Recursion Unrolling for Divide and Conquer Programs *

Radu Rugina and Martin Rinard
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139
{rugina, rinard}@lcs.mit.edu

Abstract

This paper presents *recursion unrolling*, a technique for improving the performance of recursive computations. Conceptually, recursion unrolling inlines recursive calls to reduce control flow overhead and increase the size of the basic blocks in the computation, which in turn increases the effectiveness of standard compiler optimizations such as register allocation and instruction scheduling. We have identified two transformations that significantly improve the effectiveness of the basic recursion unrolling technique. *Conditional fusion* merges conditionals with identical expressions, considerably simplifying the control flow in unrolled procedures. *Recursion re-rolling* rolls back the recursive part of the procedure to ensure that a large unrolled base case is always executed, regardless of the input problem size.

We have implemented our techniques and applied them to an important class of recursive programs, divide and conquer programs. Our experimental results show that recursion unrolling can improve the performance of our programs by a factor of between 3.6 to 10.8 depending on the combination of the program and the architecture.

1 Introduction

Iteration and recursion are two fundamental control flow constructs. Iteration repeatedly executes the loop body, while recursion repeatedly executes the body of the procedure. Loop unrolling is a classical compiler optimization. It reduces the control flow overhead by producing code that tests the loop termination condition less frequently. By textually concatenating copies of the loop body, it also typically increases the sizes of the basic blocks, improving the effectiveness of other optimizations such as register allocation and instruction scheduling.

This paper presents *recursion unrolling*, an analogous optimization for recursive procedures. Recursion unrolling uses a form of procedure inlining to transform the recursive procedures. Like loop unrolling, recursion unrolling reduces control flow overhead and increases the size of the basic blocks in the computation. But recursion unrolling is somewhat more complicated than loop unrolling. The basic form of recursion unrolling reduces procedure call overheads such as saving registers and stack manipulation. We have developed two transformations that optimize the code further. *Conditional fusion* merges conditionals with identical expressions, considerably simplifying the control flow in unrolled procedures and increasing the sizes of the basic blocks. *Recursion re-rolling* rolls back the recursive part of the procedure to ensure that a large unrolled base case is always executed, regardless of the problem size.

*This research was supported in part by NSF Grant CCR-9702297.

1.1 Divide and Conquer Programs

We have applied recursion unrolling to divide and conquer programs [10, 8, 5]. Divide and conquer algorithms solve problems by breaking them into smaller subproblems, then combining the results to generate a solution to the original problem. They use recursion as their primary control structure to generate and solve the smaller subproblems. When the division has reduced the problem to a small size, a base case computation terminates the recursion.

Divide and conquer algorithms have several appealing properties that make them a good match for modern parallel machines. First, they tend to have a lot of inherent parallelism. Once the division phase is complete, the subproblems are usually independent and can therefore be solved in parallel. Moreover, the recursive structure of the algorithm naturally leads to recursively generated concurrency. Both the divide and combine phases can execute in parallel with divide and combine phases from other subproblems. This approach typically generates more than enough concurrency to keep the machine busy.

Second, divide and conquer programs also tend to have good cache performance. Once a subproblem fits in the cache, the program reuses the cached data until the subproblem has been completely solved. Because most of the work takes place deep in the recursive call tree, the program usually spends most of its execution time running out of the cache. Furthermore, divide and conquer programs naturally work well with a range of cache sizes and at all levels of the memory hierarchy. As soon as a subproblem fits into one level of the memory hierarchy, the program runs out of that level or below until the problem has been solved. Divide and conquer programs therefore automatically adapt to different cache hierarchies, and tend to run well without modification on any machine.

To fully exploit these properties, divide and conquer programs have to be efficient and execute useful computation most of the time, rather than spending substantial time on dividing problems into subproblems, or on combining subproblems. The size of the base case controls the balance between the computation time and the divide and combine time. If the base case is too small, the program spends most of its time in the divide and combine phases instead

of performing useful computation. Unfortunately, the simplest and least error-prone coding styles reduce the problem to its minimum size (typically a problem size of one) before applying a very simple base case. Programmers therefore typically start with a simple program with a small base case, then unroll the recursion by hand to obtain a larger base case with better performance. This manual recursion unrolling is a tedious, error-prone process that obscures the structure of the code and makes the program much more difficult to maintain and modify.

The recursion unrolling algorithm presented in this paper automates the process of generating efficient base cases. It gives the programmer the best of both worlds: clean, simple divide and conquer programs with efficient execution.

1.2 Conditional Fusion

Since divide and conquer programs split the given problem into several subproblems, their recursive part typically contains multiple recursive calls. The size of the unrolled code therefore increases exponentially with the number of times the recursion is unrolled. Moreover, the control flow of the unrolled recursive procedure also increases exponentially in complexity. The typical structure of a divide and conquer program is a conditional with the base case on one branch and the recursive calls on the other branch. Recursion unrolling generates an exponential number of nested if statements. To substantially simplify the control flow in the unrolled code, we apply a transformation called *conditional fusion* which merges conditional statements with equivalent test conditions. This transformation simplifies the generated code and improves the performance by reducing the number of conditional instructions and coalescing groups of small basic blocks into larger basic blocks.

1.3 Recursion Re-Rolling

Recursion unrolling increases the code size both for the base case and for the recursive part. Compared to the recursive part of the original recursive program, the recursive part of the unrolled procedure divides the given problem into a larger number of smaller subproblems. This has the advantage that several recursive levels are removed from the recursion call tree. But this accelerated division into subproblems may generate base case subproblems of small size, even when the recursion unrolling produces unrolled base cases for larger problem sizes. To ensure that the computation always executes the more efficient larger base case, we apply another transformation, *recursion re-rolling*, which replaces the recursive part of the unrolled procedure with the recursive part of the original program.

1.4 Contributions

This paper makes the following contributions:

- **Recursion Unrolling:** It presents a new technique, recursion unrolling, for inlining recursive procedures. This technique iteratively constructs a set of unrolled recursive procedures. At each iteration, it conceptually inlines calls to recursive procedures.
- **Target programs:** It shows how to use recursion unrolling for an important class of recursive programs, divide and conquer programs. Recursion unrolling is used for these programs to automatically generate more efficient unrolled base cases of larger size.

```
void dcInc(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    dcInc(p,     n/2);
    dcInc(p+n/2, n/2);
  }
}
```

Figure 1: Divide and Conquer Array Increment Example

- **Code Transformations:** It presents two new code transformations that substantially improve the performance of recursive code resulting from recursion unrolling of divide and conquer programs. Conditional fusion reduces control flow in the unrolled code. Recursion re-rolling rolls back the recursive part of the unrolled procedure to ensure that the computation always executes large base cases. These transformations reduce control flow overhead and increase the sizes of the basic blocks.
- **Experimental Results:** It presents experimental results that characterize the effectiveness of the algorithms on a set of benchmark programs. Our results show that the proposed code transformations can substantially improve the performance of our set of divide and conquer benchmark programs.

The remainder of the paper is organized as follows. Section 2 presents a running example that we use throughout the paper. Section 3 presents the analysis algorithms. Section 5 presents experimental results from our implementation. Section 6 discusses related work. We conclude in Section 7.

2 Example

Figure 1 presents a simple example that illustrates the kinds of computations that our recursion unrolling is designed to optimize. The `dcInc` procedure implements a recursive, divide-and-conquer algorithm that increments each element of an array.

In the divide part of the algorithm, the `dcInc` procedure divides each array into two subarrays. It then calls itself recursively to increment the elements in each subarray. After the execution of several recursive levels the program generates a subarray with only one element, at which point the algorithm uses the simple base case statement `*p += 1` to directly increment the single element of the subarray.

Reducing the problem to a base case of one is the simplest way to write the program. Larger base cases require a more complex algorithm, which in general can be quite difficult to correctly code and debug. But while the small base case is the simplest and easiest way to code the computation, it has a significant negative effect on the performance — the procedure spends most of its time dividing the problem into subproblems. The overhead of control flow, consisting of procedure calls and testing for the base case condition, overwhelms the useful computation. For each instruction that increments an array element, the computation executes at least one conditional instruction and one procedure call. To improve the efficiency of the program, the compiler has to reduce the control flow overhead.

```
void dcIncI(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    if (n/2 == 1) {
      *p += 1;
    } else {
      dcIncI(p,       n/2/2);
      dcIncI(p+n/2/2, n/2/2);
    }
    if (n/2 == 1) {
      *(p+n/2) += 1;
    } else {
      dcIncI(p+n/2,       n/2/2);
      dcIncI(p+n/2+n/2/2, n/2/2);
    }
  }
}
```

Figure 2: Program After Inlining Recursive Calls

The computation of the procedure can be regarded as a sequence of useful computation instructions separated by control flow constructs. To reduce the control flow overhead the compiler must eliminate some of these control flow constructs. After removing these constructs it can then coalesce adjacent useful instructions. The resulting straight line code has less control flow overhead and provides a larger base case.

The compiler can achieve control flow elimination in two ways. Procedure inlining can eliminate procedure call overhead. Fusing conditional statements with equivalent conditional expressions can eliminate redundant conditional statements. Our compiler applies both kinds of optimizations.

2.1 Inlining Recursive Procedures

The compiler first inlines the two recursive calls to procedure `dcInc`. Figure 2 shows the result of inlining these recursive calls. For this transformation, the compiler starts with two recursive copies of the original procedure `dcInc`, replaces their recursive calls with mutually recursive calls from one copy to the other, and then inlines one of them into the other. The resulting recursive procedure `dcIncI` has two base cases: the original base case for `n = 1` and a larger base case for `n/2 = 1` (textually, this larger base case is split into two pieces in the generated code). The larger base case (for `n/2 = 1`) performs twice as much computation as the original base case (for `n = 1`).

Compared to the original recursive procedure `dcInc` from Figure 1, the inlined procedure `dcIncI` also divides the given problem into a larger number of smaller subproblems. The inlined procedure generates four subproblems of quarter size, while the original procedure generates only two problems of half size. The transformation therefore eliminates half of the procedure calls in the dynamic call tree of the program.

2.2 Conditional Fusion

The inlined code in `dcIncI` also contains more conditionals and basic blocks than the original recursive code. Since inlining has exposed more code in the body of the procedure, the compiler can now perform intra-procedural transformations to simplify the control flow of the procedure body.

```
void dcIncF(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    if (n/2 == 1) {
      *p += 1;
      *(p+n/2) += 1;
    } else {
      dcIncF(p,           n/2/2);
      dcIncF(p+n/2/2,     n/2/2);
      dcIncF(p+n/2,       n/2/2);
      dcIncF(p+n/2+n/2/2, n/2/2);
    }
  }
}
```

Figure 3: Program After Conditional Fusion

In Figure 2 the compiler recognizes that the two if statements have identical test conditions `n/2 == 1`. It therefore applies another transformation, called *conditional fusion*, to replace the two conditional statements with a single conditional statement. Figure 3 presents the resulting recursive procedure `dcIncF` after this transformation. The true branch of the new conditional statement is the concatenation of the true branches of the initial if statements. Similarly, the false branch of the new conditional statement is the concatenation of the false branches of the initial if statements. The test condition of the merged conditional statement is the common test condition of the initial if statements.

2.3 Unrolling Iterations

Because of the recursive structure of the program, the above transformations can be repeatedly applied. The recursive program `dcIncF` in Figure 3 represents the program after the first unrolling iteration. It performs the same overall computation as the original program `dcInc` from Figure 1, but it has a different internal structure.

The compiler can now use `dcIncF` and `dcInc` to unroll the recursion further. Since the two procedures perform the same computation, the compiler can safely replace their recursive calls with mutually recursive calls between each other and then inline one of them into the other. The compiler can further apply conditional fusion on the resulting recursive procedure. It thus produces the result of the second unrolling iteration, the recursive procedure `dcInc2` shown in Figure 4. It has a bigger base case for `n/2/2 == 1` and its recursive part divides the problem into an even larger number of smaller subproblems than `dcIncF`.

The recursion unrolling process can continue now by transforming recursive procedures `dcInc` and `dcInc2` into mutually recursive procedures, and then applying the above transformations. The unrolling process stops when the number of iterations reaches the desired unrolling factor.

2.4 Re-Rolling Recursion

Inlining recursive procedures automatically unrolls both the base case and the recursive part. Depending on the input problem size, the unrolled recursive part may lead to small base case subproblems that do not exercise the bigger, unrolled base cases. For instance, for procedure `dcInc2`, if the initial problem size is `n = 8`, the recursive calls will divide

```
void dcInc2(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    if (n/2 == 1) {
      *p += 1;
      *(p+n/2) += 1;
    } else {
      if (n/2/2 == 1) {
        *p += 1;
        *(p+n/2/2) += 1;
        *(p+n/2) += 1;
        *(p+n/2+n/2/2) += 1;
      } else {
        dcInc2(p,                    n/2/2/2);
        dcInc2(p+n/2/2/2,            n/2/2/2);
        dcInc2(p+n/2/2,              n/2/2/2);
        dcInc2(p+n/2/2+n/2/2/2,      n/2/2/2);
        dcInc2(p+n/2,                n/2/2/2);
        dcInc2(p+n/2+n/2/2/2,        n/2/2/2);
        dcInc2(p+n/2+n/2/2,          n/2/2/2);
        dcInc2(p+n/2+n/2/2+n/2/2/2, n/2/2/2);
      }
    }
  }
}
```

Figure 4: Program After Second Unrolling Iteration

the problem into subproblems of size `n = 1`. Therefore, the bigger base case for `n == 4` does not get executed.

Since most of the time is spent at the bottom of the recursion tree, the goal of the compiler is to ensure that the bigger base cases are always executed. To obtain this goal, the compiler applies a final transformation, called *recursion re-rolling*, which rolls back the recursive part of the unrolled procedure. The result of re-rolling procedure `dcInc2` is shown in Figure 5, in figure `dcIncR`. The compiler detects that the recursive part of the initial procedure `dcInc` is executed on a condition which is always implied by the condition on which the recursive part of the unrolled procedure `dcInc2` is executed. The compiler can therefore safely replace the recursive part of `dcInc2` with the recursive part of `dcInc`, thus rolling back only the recursive part of the unrolled procedure. Thus, the recursive part of the procedure is unrolled only temporarily, to generate the base cases. After the large base cases are generated, the recursive part is rolled back.

In the remainder of the paper, we present the algorithms that the compiler uses to perform the transformations of the program discussed in this section. We use the array increment program in Figure 1 as a running example to illustrate how our algorithms work.

3 Algorithms

This section presents in detail the algorithms that enable the compiler to perform the transformations presented in the previous section. We first present the overall algorithm, then we describe each transformation in turn.

```
void dcIncR(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    if (n/2 == 1) {
      *p += 1;
      *(p+n/2) += 1;
    } else {
      if (n/2/2 == 1) {
        *p += 1;
        *(p+n/2/2) += 1;
        *(p+n/2) += 1;
        *(p+n/2+n/2/2) += 1;
      } else {
        dcIncR(p,     n/2);
        dcIncR(p+n/2, n/2);
      }
    }
  }
}
```

Figure 5: Program After Re-Rolling

Algorithm *RecursionUnrolling* (Proc f, Int m)

$f^{(0)}_{unroll}$ = clone(f);

for (i=1; i $\leq m$; i++)
  $f^{(i)}_{unroll}$ = RecursionInline($f^{(i-1)}_{unroll}$, f);
  $f^{(i)}_{unroll}$ = ConditionalFusion($f^{(i)}_{unroll}$);

$f^{(m)}_{reroll}$ = RecursionReRolling($f^{(m)}_{unroll}$, f);

return $f^{(m)}_{reroll}$

Figure 6: Top Level of the Recursion Unrolling Algorithm

3.1 Top Level Algorithm

Figure 6 presents the top level algorithm for recursion unrolling. The algorithm takes two parameters: a recursive procedure f to unroll, and an unrolling factor m. The algorithm will unroll f m times. The algorithm iteratively builds a set $S = \{f^{(i)}_{unroll} \mid 0 \leq i \leq m\}$ of unrolled versions of the given recursive procedure. Different versions have base cases of different sizes. The internal structure of different versions is therefore different, but all versions perform the same computation.

The algorithm starts with a copy of the procedure f. This is the version of f unrolled zero times, $f^{(0)}_{unroll}$. Then, at each iteration i, the algorithm uses the version $f^{(i-1)}_{unroll}$ created in the previous iteration to build a new unrolled version $f^{(i)}_{unroll}$ of f with a bigger base case. To create the new version, the compiler inlines the original procedure f into the version from the previous iteration. The recursion inlining algorithm performs the inlining of recursive procedures. It takes two recursive versions from the set S and inlines one into another. The safety of this transformation is presented

Algorithm *RecursionInline* (Proc $f1$, Proc $f2$)

Proc $f3$ = clone($f1$);
Proc $f4$ = clone($f2$);

foreach *cstat* ∈ CallStatements($f3$, $f3$) **do**
replace callee $f3$ in *cstat* with $f4$

foreach *cstat* ∈ CallStatements($f4$, $f4$) **do**
replace callee $f4$ in *cstat* with $f3$

foreach *cstat* ∈ CallStatements($f3$, $f4$) **do**
replace *cstat* with inlined procedure $f4$

return $f3$

Figure 7: Recursion Inlining Algorithm for Two Versions of the Same Recursive Computation

in Section 4. After inlining, the compiler applies conditional fusion to simplify the control flow and coalesce conditional statements in the new recursive version $f^{(i)}_{unroll}$.

After it executes m iterations, the compiler stops the unrolling process. The last unrolled version $f^{(m)}_{unroll}$ has the biggest base case and the biggest recursive part. The compiler finally applies recursion re-rolling to roll back the recursive part of $f^{(m)}_{unroll}$.

We emphasize here that recursion inlining moves code from the inter-procedural level to the intra-procedural level and conditional fusion moves code from the inter-basic-block level to the intra-basic-block level. Both transformations give the opportunity for subsequent compiler passes to perform more aggressive optimizations.

3.2 Recursion Inlining

The recursion inline algorithm takes two recursive procedures, $f1$ and $f2$, which perform the same computation, and inlines one of them into the other. The result of this transformation is a recursive procedure with a base case bigger than any of the base cases of procedures $f1$ and $f2$.

Figure 7 presents the recursion inlining algorithm. Here, CallStatements(f, g) represents the set of procedure call statements with caller f and callee g. The compiler first creates two copies $f3$ and $f4$ of the parameter procedures $f1$ and $f2$, respectively. It then replaces each recursive call in $f3$ and $f4$ with calls to the other procedure. Because $f1$ and $f2$ perform the same computation, each of the new mutually recursive procedures $f3$ and $f4$ will perform the same computation as the original procedures $f1$ and $f2$.

With direct recursion translated into mutual recursion, each call statement has a different caller and callee. This enables procedure inlining at the mutually recursive call sites. The compiler inlines the procedure $f4$ into $f3$, replacing all call sites to $f3$ with the body of $f4$. The new version of the procedure $f3$ is a recursive procedure which performs the same computation as the given procedures $f1$ and $f2$, but has a bigger base case and splits a given problem into a larger number of smaller subproblems.

In general, procedure inlining creates new local variables for the parameters of the inlined procedure. Our compiler optimizes inlining by textually replacing actuals in the callee if the parameters are not written by the callee. This optimization relies, however, on the ability of subsequent optimization passes in the compiler to recognize common subexpressions whenever the callee reads an actual parameter multiple times.

```
void dcIncM1(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    dcIncM2(p,     n/2);
    dcIncM2(p+n/2, n/2);
  }
}

void dcIncM2(int *p, int n) {
  if (n == 1) {
    *p += 1;
  } else {
    dcIncM1(p,     n/2);
    dcIncM1(p+n/2, n/2);
  }
}
```

Figure 8: Example of Mutually Recursive Procedures

For our example, in the first unrolling iteration, the compiler performs recursion inlining with the initial procedure `dcInc` for both parameters of the inlining algorithm. The compiler first generates the mutually recursive procedures `dcIncM2` and `dcIncM1` shown in Figure 8. It then inlines procedure `dcIncM2` into `dcIncM1` at the recursive call sites `dcIncM2(p,n/2)` and `dcIncM2(p+n/2,n/2)`. It thus generates the unrolled procedure `dcIncI` from Figure 2. In the second recursion unrolling iteration, the compiler uses the initial procedure `dcInc` and the unrolled procedure `dcIncF` as parameters to the recursion inlining algorithm.

In the top level algorithm from Figure 6 the compiler could use other arguments to the recursion inlining method. Any pair of procedures in S with unrolling index smaller than i would be a valid choice for the inlining algorithm arguments. For instance, if the top level algorithm invokes the inlining method with *RecursionInline*($f^{(i-1)}_{unroll}$, $f^{(i-1)}_{unroll}$), it produces bigger base cases much faster, at a super-exponential growth rate with respect to the number of unrolling iterations. We chose a smaller increase of the base case to avoid code size blow-up and to allow the compiler to choose the best base case size from a wider spectrum of sizes.

3.3 Unrolled Code Size

Divide and conquer programs usually split the given problem into multiple subproblems. When the program has more than two recursive calls, the algorithm from Figure 6 produces an exponential increase in the code size. Let r be the number of recursive calls, C the code size of a procedure call, and B the base case code size of the initial program. Then the number of recursive calls and the program code size after i unrolling iterations, are:

$$\begin{aligned} \text{NumberRecCalls}(i) &= r^{i+1} \\ \text{CodeSize}(i) &= \frac{r^{i+1}-1}{r-1} \cdot B + r^{i+1} \cdot C \end{aligned}$$

```
Algorithm ConditionalFusion ( Proc f )

  foreach meta-basic-block B
      in bottom-up traversal of f do

    Boolean failed = false
    Statement newcond

    foreach meta-statement stat in B do
      if ( not IsConditional(stat) ) then
        failed = true
        break
      else if ( IsEmpty(newcond) )
        newcond = clone(stat)
      else if ( not SameCondition(newcond, stat) )
        failed = true
        break
      else
        Append(newcond.True, stat.True);
        Append(newcond.False, stat.False);

    if (not failed) then
      replace B with newcond

  return f
```

Figure 9: Conditional Fusion Algorithm

Also, if the initial program has one conditional statement and no conditional fusion transformation is applied, the number of conditional statements after i unrolling iterations is:

$$\text{NumberCond}(i) = \frac{r^{i+1} - 1}{r - 1}$$

Finally, if recursion inlining is invoked with $f_{unroll}^{(i-1)}$ for both arguments, the growth rate is super-exponential. The number of recursive calls after i iterations is r squared i times:

$$\text{NumberRecCalls}(i) = r^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$$

3.4 Conditional Fusion

Conditional fusion is an intra-procedural transformation that merges conditional `if` statements with equivalent condition expressions. The conditional fusion algorithm searches the control flow graph of the unrolled procedure for consecutive conditional statements with this property.

For detecting such patterns, a *hierarchically* structured control flow graph is more appropriate. A hierarchical control flow graph is a graph of *meta-basic-blocks*. A meta-basic-block is a sequence of *meta-statements*. A meta-statement is either a program instruction, a conditional statement, or a loop statement. There is no program instruction that jumps in or out of a meta-basic-block. Bodies of loop statements and branches of conditional statements are, in turn, hierarchical control flow graphs. Flattening the hierarchical control flow graph of a procedure produces the (regular) control flow graph of that procedure.

Using a hierarchical control flow graph representation, the conditional fusion algorithm is formulated as shown in Figure 9. The compiler traverses the hierarchical control flow structure in a bottom-up fashion. At each level, it inspects the meta-statements in the current basic block B. It checks if all the meta-statements in B are conditional statements and if they all have equivalent condition expressions. If not, the *failed* flag is set to true and no transformation is performed. When checking the equivalence of condition expressions, the compiler also verifies that the conditional statements do not write any of the variables of the condition expressions. This ensures that condition expressions of different if statements refer to variables with the same values. As it checks the statements, the compiler starts building the merged `if` statement. If *stat* is the current conditional statement, the compiler appends its true branch to the true branch of the new conditional, and its false branch to the false branch of the new conditional. After scanning the whole basic block, if the flag *failed* is not set, the compiler replaces B with the newly constructed conditional statement.

The algorithm could be extended to duplicate instructions between conditional with equivalent test expressions, or to partially merge conditionals whose test expressions are equivalent only for some values of the program variables. But for our benchmarks, the simple version of loop fusion from Figure 9 is enough to substantially simplify the control flow after recursion inlining.

Given a divide and conquer procedure with a single conditional statement and r recursive calls, conditional fusion can potentially reduce the number of conditional statements after i recursive unrolling iterations from r^i to only i conditionals.

```
Algorithm RecursionReRoll ( Proc f1, Proc f2 )

  MetaBasicBlock B1 = RecursivePart(f1)
  MetaBasicBlock B2 = RecursivePart(f2)

  Boolean cond1 = RecursionCondition(f1)
  Boolean cond2 = RecursionCondition(f2)

  if ( cond1 implies cond2 ) then
    replace calls in B2 to f1 with calls to f2
    replace B1 with B2 in procedure f1

  return f1
```

Figure 10: Recursion Re-Rolling Algorithm for Two Versions of the Same Recursive Computation

3.5 Recursion Re-Rolling

The recursion re-rolling transformation rolls back the recursive part of the unrolled procedure, leaving the unrolled base case unchanged. It ensures that the largest unrolled base case is always executed, regardless of the input problem size.

Figure 10 presents the algorithm for recursion re-rolling. The algorithm is given two procedures which are versions of the same recursive computation. Procedure $f1$ has an unrolled recursive part and procedure $f2$ has a rolled recursive part. To re-roll the recursion of $f1$, the compiler first identifies the recursive parts two procedures, $B1$ and

$B2$ respectively. The recursive part of a procedure is the smallest meta-basic-block in the procedure that contains all the recursive calls and which represents the whole body of the procedure when executed.

The compiler then detects the conditions on which the recursive parts are executed. If the condition *cond*1 on which the recursive part of the unrolled procedure is executed always implies the condition *cond*2 on which the rolled recursion of $f2$ is executed, then the compiler performs the re-rolling transformation. Again, the compiler verifies that variables in conditional expressions are not written by other statements. It therefore ensures that variables in different conditions refer to the same values. Knowing that both $f1$ and $f2$ perform the same computation, the compiler first replaces calls to $f2$ in $B2$ with calls to $f1$. It then replaces block $B1$ with block $B2$ to complete rolling back the recursive part of $f1$.

In our example, the recursion condition of the unrolled procedure `dcInc2` from Figure 4 is *cond*1: $(\texttt{n} \neq \texttt{1}) \wedge (\texttt{n/2} \neq \texttt{1}) \wedge (\texttt{n/2/2} \neq \texttt{1})$. The recursion condition of the initial procedure `dcInc` is *cond*2: $\texttt{n} \neq \texttt{1}$. Obviously, *cond*1 always implies *cond*2, so the compiler replaces the recursive part of `dcInc2` with the rolled back recursive part of `dcInc`. The resulting procedure with unrolled base case and re-rolled recursion is procedure `dcIncR` from Figure 5.

4 Correctness of Transformations

In this section, we present theorems that guarantee the correctness of the algorithms presented in the paper. To simplify the presentation, we assume that procedures do not access local variables from other procedures and that recursive procedures do not have any call sites except for the recursive call sites. We also assume that all the procedures have the same signature, i.e., the same number of arguments and the same argument types.

4.1 Proof Concepts

We define the following concepts:

- *Program State:* The *program state* for a procedure is the set of values for global variables, procedure parameters, and heap-allocated variables.
- *Procedure Activations:* A *procedure activation* $A(f, s_0)$ is an instance of the procedure f with input program state s_0. For an activation $A(f, s_0)$ whose execution terminates, $O(f, s_0)$ denotes the program state after the exe3cution of that activation.
- *Dynamic Call Trees:* $T(f, s_0)$ denotes the *dynamic call tree* rooted at an activation $A(f, s_0)$. The nodes in $T(f, s_0)$ correspond to procedure activations invoked during the execution of $A(f, s_0)$. Also, $H(f, s_0)$ denotes the height of the call tree $T(f, s_0)$. If the execution of an activation produces an infinite call chain, the height of the call tree is infinite. For activations whose execution terminates, the call tree is finite. There may be activations that do not terminate, but have finite dynamic call trees, as in the case of intra-procedural infinite loops.
- *State Sets:* Given a directly or indirectly recursive procedure f and an input program state s_0, we define the *state set* $S(f, s_0)$ to be the set of all input states in all activations in $T(f, s_0)$. We define the *call site state set* $C(f, s_0)$ to be the set of all input states in activations directly invoked by $A(f, s_0)$. We also define the *terminating state set* $Term(f)$ to be the set of input states on which the procedure f terminates.
- *Terminating Equivalent Procedures:* Given two recursive procedures f and g, we say that f and g are *terminating equivalent*, written as $f \equiv_t g$, if $Term(f) = Term(g)$.
- *Semantically Equivalent Procedures:* Given two recursive procedures f and g, we say that f and g are *semantically equivalent* or *perform the same computation*, written as $f \equiv_s g$, if $f \equiv_t g$ and for any input state $s_0 \in Term(f)$ the execution of the activations $A(f, s_0)$ and $A(g, s_0)$ yields the same output program state $O(f, s_0) = O(g, s_0)$, after the execution of the procedures.
- *Recursively Included Procedures:* Given two recursive procedures f and g, we say that f is *recursively included* in g, written as $f \preceq_r g$, if $f \equiv_t g$ and for any input state $s_0 \in Term(f)$ and any call site state $s_1 \in C(f, s_0)$ we have $s_1 \in S(g, s_0)$. Conceptually, the program state at each call site in $A(f, s_0)$ is the input state of an activation in the call tree $T(g, s_0)$, at depth greater than one. This means that f recurses at least as fast as g.

To prove the transformations correct, we need to prove that they generate new procedures that are semantically equivalent to the original procedure. The proofs of correctness for the traditional inlining and conditional fusion transformations are straightforward, and we omit these proofs.

4.2 Correctness of Recursion Inlining

For the recursion inlining transformation, the key is to show that transforming direct recursion to mutual recursion preserves both recursive inclusion and semantic equivalence.

Given two recursive procedures $f1$ and $f2$, we denote by ($f3$,$f4$) = MutuallyRecursive($f1$,$f2$) the pair of mutually recursive procedures $f3$ and $f4$ generated from $f1$ and $f2$ as follows. The transformation first clones $f1$ and $f2$, calling the clones $f3$ and $f4$. It then replaces each call in $f3$ to $f1$ with a call to $f4$, and each call in $f4$ to $f2$ with a call to $f3$. Note that except for call statements, the procedures $f1$ and $f3$ are identical, as are the procedures $f2$ and $f4$. The recursion inlining algorithm from Figure 7 basically constructs the pair ($f3$,$f4$), and then inlines one procedure into the other. The following theorem shows that the constructed pair of mutually recursive procedures ($f3$,$f4$) preserves recursive inclusion and semantic equivalence to the original procedure.

Theorem 1 *Let f, $f1$ and $f2$ be recursive procedures such that:*

$$f1 \preceq_r f \quad \textit{and} \quad f1 \equiv_s f$$
$$f2 \preceq_r f \quad \textit{and} \quad f2 \equiv_s f$$

If ($f3$,$f4$) = MutuallyRecursive($f1$,$f2$) then:

$$f3 \preceq_r f \quad \textit{and} \quad f3 \equiv_s f$$
$$f4 \preceq_r f \quad \textit{and} \quad f4 \equiv_s f$$

The proof is by induction on the height of dynamic call subtrees for terminating procedure activations. We present this proof in Appendix A.1. This theorem, combined with the fact that the standard inlining transformation preserves recursive inclusion and semantic equivalence, enables us to conclude that the procedure returned by the algorithm in Figure 7 is recursively included in and semantically equivalent to the original procedure.

4.3 Correctness of Unrolled Versions

We next use Theorem 1 to prove that the unrolled versions are recursively included in and semantically equivalent to the original recursive procedure f.

Theorem 2 *Let f be a recursive procedure. Then the unrolled versions $f^{(i)}_{unroll}$ from Figure 6 satisfy the following relations:*

$$\forall i \geq 0 : \; f^{(i)}_{unroll} \preceq_r f \;\; and \;\; f^{(i)}_{unroll} \equiv_s f$$

The proof is by induction on the iteration index i. The proof directly comes from Theorem 1 combined with the fact that the conditional fusion transformation preserve recursive inclusion and semantic equivalence.

4.4 Correctness of Recursion Re-Rolling

The following theorem guarantees that the recursion re-rolling algorithm constructs a procedure that is semantically equivalent to the original recursive procedure.

Theorem 3 *Let f, $f1$ and $f2$ be recursive procedures such that:*

$$f1 \equiv_s f$$
$$f2 \equiv_s f$$

If $f3$ = RecursionReRoll($f1$,$f2$), then:

$$f3 \equiv_s f$$

The proof is presented in Appendix A.2. It is based on the observation that $f3$ always recurses exactly as $f2$ up to a certain input state, at which point it executes a base case computation identical to the base case computation of $f1$ with the that same input. Finally, Theorem 2 combined with Theorem 3 guarantees that the overall recursion unrolling algorithm from Figure 6 constructs a recursive procedure that is semantically equivalent to the original recursive procedure.

5 Experimental Results

We used the SUIF compiler infrastructure [1] to implement the recursion unrolling algorithms presented in this paper. We present experimental results for two divide and conquer programs:

- **Mul:** Divide and conquer blocked matrix multiply. The program has one recursive procedure with 8 recursive calls and a base problem size of a matrix with one element.
- **LU:** Divide and conquer LU decomposition. The program has four mutually recursive procedures. Each of them has a base problem size of a matrix with one element. The main recursive procedure has 8 recursive calls.

We implemented our compiler as a source-to-source translator. It takes a C program as input, locates the recursive procedures, then unrolls the recursion to generate a new C program. We then compiled and ran the generated C programs on three machines:

- **Pentium III:** A Pentium III machine running Linux.
- **PowerPC:** A PowerPC running Linux.
- **Origin 2000:** An SGI Origin 2000 running IRIX.

Table 1 presents the running times for various versions of the Mul program. Each column is labeled with the number of times that the compiler unrolled the recursion; we report results for the computation unrolled 0, 1, 2, and 3 times. If the column is labeled with an f, it indicates that compiler applied the conditional fusion transformation. If the column is labeled with an r, it indicates that the compiler applied the recursion re-rolling transformation. So, for example, the column labeled 1u+f+r contains experimental results for the version with the recursion unrolled once and with both conditional fusion and recursion re-rolling. Depending on the architecture, the best automatically unrolled version of program Mul performs between 3.6 to 10.8 times better than the unoptimized version Table 2 presents the running times for various versions of the LU decomposition program. Depending on the architecture, the best automatically unrolled version of this program performs between 3.8 to 8.6 times better than the unoptimized version.

We also evaluate our transformations by comparing the performance of our automatically generated code with that of several versions of the programs with optimized, hand coded base cases. We obtained these versions from the Cilk benchmark set [9]. The last column in Tables 1 and 2 presents the running times of the hand coded versions. The best automatically unrolled version of Mul performs between 2.2 and 2.9 worse than the hand optimized version. The performance of the best automatically unrolled version of LU is basically comparable to that of the hand coded version. These results show that our transformations can generate programs whose performance is close to, and in some cases identical to, the performance of programs with hand coded base cases.

5.1 Impact of Re-Rolling

The running times in Tables 1 and 2 emphasize the impact of recursion re-rolling on the performance of the program. Whenever the unrolled recursion makes the program skip its largest unrolled base case, recursion re-rolling can deliver substantial performance improvements.

For instance, in the case of program Mul with recursion unrolled twice, running on the Origin 2000, on a matrix of 512x512 elements, recursion re-rolling dramatically improves the performance — with recursion re-rolling, the running time drops from 29.81 seconds to 3.95 seconds. For this example, recursion inlining and conditional fusion produce additional base cases of sizes 2 and 4. But because the unrolled recursive part divides each problem in four subproblems at each step, these base cases never get executed. The program always executes the inefficient base case of size 1. Re-rolling

Machine	Input Size	Unrolling Types								Hand Coded
		0u	1u	1u+f	1u+f+r	2u	2u+f	2u+f+r	3u+f+r	
Pentium III	512	9.22	3.41	2.83	2.97	11.49	9.34	2.69	2.55	1.16
Pentium III	1024	73.80	69.43	64.75	23.61	32.51	24.63	20.70	20.47	9.19
PowerPC	512	14.35	4.19	3.28	2.89	17.32	25.19	1.63	1.33	0.59
PowerPC	1024	114.60	136.84	137.20	23.17	33.87	35.91	12.91	10.74	4.69
Origin 2000	512	30.57	9.92	6.77	6.84	30.54	29.81	3.95	3.62	1.24
Origin 2000	1024	244.44	239.64	230.43	54.59	81.13	58.55	31.46	28.73	9.90

Table 1: Running Times of Unrolled Versions of Mul (seconds)

Machine	Input Size	Unrolling Types								Hand Coded
		0u	1u	1u+f	1u+f+r	2u	2u+f	2u+f+r	3u+f+r	
Pentium III	512	3.14	2.15	2.00	0.99	1.86	1.32	0.84	0.83	0.71
Pentium III	1024	24.77	14.62	13.14	8.53	21.25	15.86	6.99	6.58	5.48
PowerPC	512	4.88	4.15	4.01	1.16	2.41	2.25	0.73	0.66	0.70
PowerPC	1024	39.16	26.58	24.91	9.46	29.20	32.33	5.91	5.33	5.64
Origin 2000	512	10.77	7.34	6.56	2.42	5.06	3.31	1.39	1.26	1.20
Origin 2000	1024	85.86	48.47	40.98	19.20	54.56	44.53	10.96	9.95	9.57

Table 2: Running Times of Unrolled Versions of LU (seconds)

the recursion ensures that the efficient base case of size 4 gets always executed.

The structure of the inlined recursion also explains why programs whose recursive part is not re-rolled may perform worse after several inlining steps. For instance, the Mul program on a Pentium III has a surprisingly high running time of 11.49 seconds after two unrolling iterations, compared to its fast execution of 3.41 seconds after a single unrolling iteration. The reason for this performance difference is that, for the given problem size of 512, the problem is always reduced to a base case of size 2 when recursion is unrolled once, while the program always ends up executing the smaller and inefficient base case of size 1 when recursion is unrolled twice. Re-rolling the recursion ensures that the largest and most efficient base case is always executed. For example, consider Mul with the recursion unrolled twice, running on the Pentium III, on a matrix of 512x512 elements. The running time after re-rolling drops to 2.69 seconds as compared to 9.34 seconds without recursion unrolling.

Finally, our results for different program sizes show that the impact of re-rolling depends on the program size. For LU running on the PowerPC, with recursion unrolled twice, the version with re-rolling runs 3.08 times faster than the original version for a matrix of 512x512 elements, and 5.47 times faster than the original version for a matrix of 1024x1024 elements. The fact that the size of the base case that gets executed before re-rolling depends on the problem size explains this discrepancy. Our results show that in the vast majority of cases, re-rolling improves the performance regardless of the input problem size. The only two exceptions are for program Mul with one unrolling iteration, running on a matrix of 512x512 elements. Here, re-rolling produces a slight slowdown because the programs were already executing their largest base cases before re-rolling.

5.2 Impact of Conditional Fusion

Our results show that conditional fusion can achieve speedups of up to 1.5 over the versions without conditional fusion, as in the case of the Mul program running on the SGI Origin 2000, on a matrix of 512x512 elements, with recursion unrolled once. In the majority of cases, fusion of conditionals improves program performance. In a few cases, though, the modified cache behavior after conditional restructuring causes a slight degradation in the performance.

The major advantage of conditional fusion transformation is that it enables recursion re-rolling, which has significant positive impact on the running times. To apply recursion re-rolling, the compiler has to identify and separate the recursive parts and the base cases. Conditional fusion is the key transformation that enables the compiler to identify these parts in the unrolled code.

Finally, for all our benchmarks, the combined effects of conditional fusion and recursion re-rolling always improves the running times as compared with the programs with recursion unrolling alone.

6 Related Work

Procedure inlining is a classical compiler optimization [3, 2, 4, 6, 12, 7, 11]. The usual goal is to eliminate procedure call and return overhead and to enable further optimizations by exposing the combined code of the caller and callee to the intraprocedural optimizer. Some researchers have reported a variety of performance improvements from procedure inlining; others have reported that procedure inlining has relatively little impact on the performance.

Our initial recursion unrolling transformation is essentially procedure inlining. We augment this transformation with two additional transformations, conditional fusion and recursion re-rolling, that significantly improve the performance of our target class of divide and conquer programs. We therefore obtain the benefit of a reduction in procedure call and return overhead. We also obtain more efficient code that eliminates redundant conditionals and sets up the recursion so as to execute the efficient large base case most of the time.

In general, we report much larger performance increases than other researchers. We attribute these results to several factors. First, we applied our techniques to programs that heavily use recursion and therefore suffer from significant overheads that recursion unrolling can eliminate. Second, conditional fusion and recursion re-rolling go beyond the standard procedure inlining transformation to further optimize the code.

7 Conclusion

This paper presents recursion unrolling, a technique for improving the performance of recursive computations. Like loop unrolling, recursion unrolling reduces control flow overhead and increases optimization opportunities by generating larger basic blocks. But recursion unrolling is somewhat more complicated than loop unrolling. The basic form of recursion unrolling reduces procedure call overheads such as saving registers and stack manipulation. We have developed two transformations that optimize the code further. *Conditional fusion* merges conditionals with identical expressions, considerably simplifying the control flow in unrolled procedures. *Recursion re-rolling* rolls back the recursive part of the procedure to ensure that the biggest unrolled base case is always executed, regardless of the problem size.

We have implemented our techniques and applied them to an important class of recursive programs, divide and conquer programs. Our experimental results show that recursion unrolling can substantially improve the performance of these programs. Specifically, they show that our combined techniques can improve the performance of our benchmark programs by a factor of between 3.6 to 10.8 depending on the combination of the program and the architecture. They also show that conditional fusion and recursion re-rolling have a significant positive effect on the overall performance of the programs.

References

[1] S. Amarasinghe, J. Anderson, M. Lam, and A. Lim. An overview of a compiler for scalable parallel machines. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[2] Andrew W. Appel. Unrolling recursion saves space. Technical Report CS-TR-363-92, Princeton University, March 1992.

[3] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989. ACM, New York.

[4] P. Chang, S. Mahlke, W. Chen, and W. Hwu. Profile-guided automatic inline expansion for C programs. *Software—Practice and Experience*, 22(5):349–369, May 1992.

[5] S. Chatterjee, A. Lebeck, P. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. In *Proceedings of the 11th Annual ACM Symposium on Parallel Algorithms and Architectures*, Saint Malo, France, June 1999.

[6] K. Cooper, M. W. Hall, and L. Torczon. An experiment with inline substitution. *Software—Practice and Experience*, 21(6):581–601, June 1991.

[7] J. W. Davidson and A. M. Holler. A study of a C function inliner. *Software—Practice and Experience*, 18(8):775–790, August 1988.

[8] J. Frens and D. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, NV, June 1997.

[9] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the SIGPLAN '98 Conference on Program Language Design and Implementation*, Montreal, Canada, June 1998.

[10] F. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.

[11] S. Richardson and M. Ganapathi. Interprocedural analysis versus procedure integration. *Information Processing Letters*, 32(3):137–142, August 1989.

[12] R. Scheifler. An analysis of inline substitution for a structured programming language. *Commun. ACM*, 20(9), September 1977.

A Correctness Proofs

A.1 Proof of Theorem 1

The proof consists of two steps. First we prove that for each input state on which procedure f terminates, procedures $f3$ and $f4$ also terminate, yield the same output state as f, and satisfy the recursive inclusion condition. Second, we prove that for each input state on which procedure $f3$ or procedure $f4$ terminates, procedure f also terminates. The combined results of these steps immediately prove Theorem 1.

Step 1. First we show that for any state $s \in Term(f)$ the following relations hold:

$$s \in Term(f3) \quad (1)$$
$$s \in Term(f4) \quad (2)$$
$$\forall s_1 \in C(f3, s): \ s_1 \in S(f, s) \quad (3)$$
$$\forall s_1 \in C(f4, s): \ s_1 \in S(f, s) \quad (4)$$
$$O(f3, s) = O(f, s) \quad (5)$$
$$O(f4, s) = O(f, s) \quad (6)$$

To prove this, we show that for any state $s' \in S(f, s)$ the above relations hold. We prove this by induction on the height h of the subtree $T(f, s')$. The proof by induction is applicable here because for any h, $0 \leq h \leq T(f, s)$, there is at least one state in $S(f, s)$ whose subtree has height h. This holds because $A(f, s)$ terminates, so $T(f, s)$ has finite height.

For the base case, let $s' \in S(f, s)$ be a program state such that $H(f, s') = 0$. Hence the activation $A(f, s')$ does not invoke any procedure. Therefore $C(f1, s') = C(f2, s') = \emptyset$, otherwise it would contradict the hypothesis that $f1 \preceq_r f$ and $f2 \preceq_r f$. It means that procedures $f1$ and $f2$ are not executing any recursive code either. But from construction, procedures $f1$ and $f3$, respectively $f2$ and $f4$, are identical except for the recursive calls. Hence activations $A(f1, s')$ and $A(f3, s')$, respectively $A(f2, s')$ and $A(f4, s')$, execute identical code, and relations (1) through (6) trivially follow the similar relations for $f1$ and $f2$ from hypothesis.

For the induction step, assume that relations (1)-(6) hold for all the states in $s' \in S(f, s)$ with $H(f, s') \leq h - 1$. We want to prove that these relations hold for all the states with recursive trees with height $H(f, s') = h$. Let s' be such a state with $H(f, s') = h$ and consider now the activations of $f1$, $f2$, $f3$, and $f4$ with input state s'. The main idea is to prove that the activations $A(f1, s')$ and $A(f3, s')$, respectively $A(f4, s')$ and $A(f2, s')$ execute the same sequence of statements and all the executed call statements terminate and yield the same program state. We show the proof only for $A(f1, s')$ and $A(f3, s')$. The proof is similar for $A(f2, s')$ and $A(f4, s')$.

First we match statements in $f1$ and $f3$. Recursive calls in $f1$ are matched with mutually recursive calls in $f3$, and all the other statements in $f1$ are matched with their identical corresponding statements from $f3$. Consider two matching statements *stmt* in $f1$ and *stmt'* in $f3$. Assume that *stmt* is reachable with input state s'' in activation $A(f1, s')$, and *stmt'* is also reachable with the same input state s'' in activation $A(f3, s')$. We want to prove that these statements terminate their execution, and the program state after the execution of the statements is the same. First consider the case when *stmt* and *stmt'* are not recursive calls. These statements are then identical, by construction. They are not call statements, therefore they both terminate. Being identical, the program state after the execution of these statements is identical. Consider now that *stmt* is a recursive call in $f1$ and *stmt'* is a mutually recursive call in $f3$ to $f4$. Since the program point before *stmt'* is reachable and is a recursive call site to $f1$, it means that $s'' \in C(f1, s')$. From hypothesis, $f1 \preceq_r f$, therefore $s'' \in C(f1, s')$ implies $s'' \in S(f, s')$. Also since $A(f1, s')$ terminates, states s' and s'' are different. Hence, $A(f, s'')$ is part of $T(f, s')$, but is not the root of the tree, so $H(f, s'') < H(f, s') = h$. Therefore we can apply the induction hypothesis for activations with input state s''. First, the induction hypothesis from rule (2) shows that $s'' \in Term(f4)$. Also, $s' \in Term(f)$, hence $s' \in Term(f1)$. This implies $s'' \in Term(f1)$. Thus, both statements *stmt* and *stmt'* terminate on input s''. Moreover, rules (5) and (6) in the induction hypothesis show that $O(f1, s'') = O(f3, s'')$. This proves that the states after the execution of statements *stmt* and *stmt'* are identical in this case as well.

Hence we proved that activations $A(f1, s')$ and $A(f3, s')$, respectively $A(f2, s')$ and $A(f4, s')$, execute the same sequence of matching statements, and the execution of each statement terminates and yields the same state. The equivalence of executions for these activations directly prove relations (1)-(6). They follow from similar relations for $f1$ and $f2$ in the hypothesis.

Step 2. We prove now that for any state $s \in Term(f3)$ we have:

$$s \in Term(f) \quad (7)$$
$$O(f, s) = O(f3, s) \quad (8)$$

and for any state $s \in Term(f4)$ we have:

$$s \in Term(f) \quad (9)$$
$$O(f, s) = O(f4, s) \quad (10)$$

We will prove only relations (7) and (8), for program states $s \in Term(f3)$. The proof for relations (9) and (10), for states $s \in Term(f4)$, is similar. Consider a state $s \in Term(f3)$. The dynamic call tree $T(f3, s)$ has therefore finite height. We will prove that for any state $s' \in S(f3, s)$ the following property holds. If s' is the input state for an activation $A(f3, s')$ then:

$$s' \in Term(f) \quad (11)$$
$$O(f, s') = O(f3, s') \quad (12)$$

and if s' is the input state for an activation $A(f4, s')$ then:

$$s' \in Term(f) \quad (13)$$
$$O(f, s') = O(f4, s') \quad (14)$$

We will prove these relations by induction on the height of the dynamic call tree of activations with input state s' in $T(f3, s)$. Again, the applicability of a proof by induction comes from the fact that $A(f3, s)$ terminates. The proof uses the same kind of reasoning as the induction proof from Step 1.

For the base case, let s' be the input state of an activation of $f3$ or of $f4$ in $T(f3, s)$, with dynamic call tree of height $h = 0$. Suppose this is an activation $A(f3, s')$ of procedure $f3$. Since $H(f3, s') = 0$, it means that this activation does not execute any recursive calls. But from construction, $f3$ is identical to $f1$ except for the procedure calls. Therefore, in this case, $f3$ and $f1$ execute identical code for input state s'. Hence, $s' \in Term(f3)$ implies $s' \in Term(f1)$. Then, using the hypothesis, $f1 \equiv_t f$, we get that $s' \in Term(f)$, which proves relation (11). Also, since $f3$ and $f1$ execute identical in this case, it means that the program state after the execution of these procedures is identical: $O(f1, s') = O(f3, s')$.

From hypothesis $f1 \equiv_s f$, therefore $O(f, s') = O(f3, s')$, which proves relation (12). We use a similar reasoning to prove relations (13) and (14) for an activation of $f4$ with height $H(f4, s') = 0$.

For the induction step, assume properties (11)-(14) hold for all the input states to activations having dynamic call trees with heights less or equal to $h-1$. Consider an activation with input state s' whose dynamic call tree has height h. Assume this is an activation $A(f3, s')$ of procedure $f3$. The proof is similar for activations of procedure $f4$. We have now to prove relations (11) and (12). Again, we first match program points and statements in $f1$ and $f3$ and prove equivalence of states at matching program points. The key to proving the induction step is to prove that reachable recursive calls in $f1$ terminate and yield the same program state as their equivalent mutually recursive calls in $f3$. Consider two matching statements *stmt* and *stmt'* such that *stmt* is a recursive call in $f1$ and *stmt'* is a mutually recursive call in $f3$ to $f4$. Let s'' be an input state to both of these statements. Since the activation $A(f4, s'')$ is part of the dynamic call tree $T(f3, s')$, if means that $H(f4, s'') \leq h-1$. Therefore, by induction hypothesis, from relation (13) for state s'', we have that $s'' \in Term(f)$. This implies, by hypothesis, that $s'' \in Term(f1)$. Therefore both *stmt* and *stmt'* terminate. Moreover, by induction hypothesis, relation (14) for s'' shows that $O(f, s'') = O(f4, s'')$. But from hypothesis, $O(f1, s'') = O(f, s)$. Hence, $O(f1, s'') = O(f4, s'')$, and the execution of statements *stmt* and *stmt'* yields the same state. Therefore, we conclude that the activations $A(f1, s')$ and $A(f3, s')$ execute the same sequence of matching statements and all of the statements in the sequence terminate and generate identical output state. Hence, $A(f1, s')$ terminates, so $s' \in Term(f1)$. In turn, this implies $s' \in Term(f)$, which proves relation (11). Equivalence of executions for $A(f1, s')$ and $A(f3, s')$ also means that $O(f3, s') = O(f1, s')$. From hypothesis, $O(f1, s') = O(f, s')$. Therefore $O(f, s') = O(f3, s')$, which proves relation (12). This concludes the induction step.

A.2 Proof of Theorem 3

The proof is based on the following observations. First, for all the program states s for which $A(f3, s)$ executes its recursive part, $A(f2, s)$ also executes its recursive part, and for all the program states s for which $A(f2, s)$ executes its base case, $A(f2, s)$ also executes its base case. By construction, $A(f2, s)$ and $A(f3, s)$ execute identical code in these cases. Second, an activation $A(f3, s)$ executes its base case if and only if the activation $A(f1, s)$ executes its base case. By construction, in this case $A(f1, s)$ and $A(f3, s)$ execute identical code. These properties directly come from the construction of $f3$.

We first extend the notion of semantic equivalence for procedure activations. We say that two activation frames $A(f, s)$ and $A(g, s')$ are semantically equivalent if $s = s'$, $s \in Term(f)$ if and only if $s \in Term(g)$, and $s \in Term(f)$ implies $O(f, s) = O(g, s)$. Basically, two procedures are semantically equivalent if all of their activations with identical input state are semantically equivalent.

The proof consists of two steps. First we prove that for each input state s on which $A(f3, s)$ terminates, $A(f2, s)$ and $A(f3, s)$ are semantically equivalent. Second, we prove that for each input state s on which $A(f3, s)$ terminates, $A(f2, s)$ and $A(f3, s)$ are semantically equivalent. The combined results of these steps immediately prove Theorem 3.

Step 1. Let $s \in Term(f3)$. We want to prove that $A(f2, s)$ and $A(f3, s)$ are semantically equivalent. Since $s \in Term(f3)$, the dynamic call tree $T(f3, s)$ is finite. From the first observation, the activations corresponding to internal nodes of the tree perform the same intra-procedural computation as the activations of $f2$ with the same input state.

Also, each leaf of $T(f3, s)$ corresponds to an activation $A(f3, s')$. From the first observation, activation $A(f3, s')$ is semantically equivalent to activation $A(f1, s')$. But from hypothesis, $f1 \equiv_s f2$, activation $A(f3, s')$ is semantically equivalent to activation $A(f2, s')$. Therefore we can replace each leaf node corresponding to an activation $A(f3, s')$ in $T(f3, s)$ with the dynamic tree of $A(f2, s')$. The resulting computation is guaranteed to terminate and is semantically equivalent to $A(f3, s)$. But this dynamic tree has identical structure with the dynamic call tree of $A(f2, s)$, and the activations corresponding to similar nodes in these trees execute the same intra-procedural computation. We therefore conclude that in this case $A(f2, s)$ and $A(f3, s)$ are semantically equivalent.

Step 2. Let $s \in Term(f2)$. We want to prove that $A(f2, s)$ and $A(f3, s)$ are semantically equivalent. Since $s \in Term(f2)$, the dynamic call tree $T(f2, s)$ is finite. For each $s' \in S(f2, s')$ such that $A(f1, s')$ executes its base case, we replace the subtree of the activation $A(f2, s')$ with a leaf corresponding to $A(f1, s')$. Because of semantic equivalence of $f1$ and $f2$, the semantics of the overall computation $A(f2, s)$ is preserved in the resulting dynamic tree. Using the two observations, we deduce that the resulting tree has the same structure as the dynamic tree of $A(f3, s)$, and the activations corresponding to similar nodes in these trees execute the same intra-procedural computation. This proves that $A(f2, s)$ and $A(f3, s)$ are semantically equivalent.