Static Elaboration of Recursion for Concurrent Software

Stephen A. Edwards [*]   Jia Zeng
Columbia University, New York
{sedwards,jia}@cs.columbia.edu

Abstract

Unlike sequential software, concurrent software needs a structuring mechanism capable of specifying constructs such as pipelines, scatter-gather, and other networks. Concurrent software languages usually provide mechanisms for dynamically creating such structures, but this makes them difficult to analyze statically. In particular, it would be very convenient to be able to put bounds on the resources (memory, processes) required by a particular system.

We present a static elaboration technique that can remove bounded recursion from concurrent programs, useful for tools that cannot handle recursion such as those for formal verification and hardware synthesis. We work with SHIM, a concurrent language that provides concurrent, recursive function calls that can be used to elegantly construct concurrent structures such as pipelines and the FFT.

Our technique first slices the program from every recursive call to determine which variables control the recursion, then perform a simulation of the program with respect to those variables, performing constant propagation to produce simplified code without recursion. Experimental results suggest our slicing procedure is effective at selecting just those variables that participate in the recursion and that our simulation technique for generating non-recursive code can quickly produce small non-recursive programs, making this technique practical.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages—Processors

General Terms Languages, Algorithms

Keywords Recursion, Static Elaboration, Partial Evaluation, Concurrency, SHIM

1. Introduction

Any big system is assembled from pieces, but the assembly problem can be much richer for parallel systems. Digital transistor circuits are an extreme example: they only contain two kinds of pieces (n- and p-channel transistors); all their functionality arises from how collections of these pieces are connected. Sequential software programs also assemble things from pieces (e.g., expressions from identifier and operators, functions from sequences of statements), but the pieces tend to be more powerful and the connection schemes more simple. Parallel software needs both: the ability to specify traditional program-like sequential behavior and the ability to specify complicated parallel structures.

```
void fifo3(chan int i,
           chan int &o) {
  fifo(i, o, 3);
}

void fifo(chan int i,
          chan int &o,
          int n) {
  if (n > 1) {
    chan int c;
      buf(i, c);
    par
      fifo(c, o, n-1);
  } else
    buf(i, o);
}

void buf(chan int i,
         chan in &o) {
  for (;;)
    next o = next i;
}
```

(a)

```
void fifo3(chan int i,
           chan int &o) {
  chan int c1, c2, c3;
    buf(i, c1);
  par
    buf(c1, c2);
  par
    buf(c2, o);
}

void buf(chan int i,
         chan in &o) {
  for (;;)
    next o = next i;
}
```

(b)

Figure 1. Removing recursion from (a) a parametric, recursive FIFO in SHIM gives (b).

In this paper, we present a technique that can remove the recursion from a parallel program that expresses its structure with bounded recursion. This is useful when working with tools that cannot accept recursion, such as certain formal verifiers and hardware compilers. Figure 1 shows a simple parametric FIFO transformed into recursion-free code. A form of partial evaluation, our procedure first slices the program at recursive function calls to understand what variables control the recursion, then statically simulates the program on those variables, constructing a simpler program along the way. It allows a programmer to use the power of recursion to describe parallel structures yet have a "static" program at the end, i.e., with no cycles in its call graph.

Our goal is to remove recursion while increasing the size of the program as little as possible. Although our static simulation program is analagous to the subset construction algorithm for converting nondeterministic finite automata to deterministic ones, it is even more likely to produce an exponential blow up in code size because it considers integer-valued variables. As a result, we are very careful to restrict the scope of our simulation procedure to only a few variables. Since our goal is to remove recursion, we first use a slic-

[*] Edwards and Zeng are supported by the NSF, Intel, Altera, the SRC, and NYSTAR.

```
[10] CHAN BYTE reg :
PAR process = 0 FOR 7
  WHILE TRUE
    BYTE data :
    SEQ
      reg[process] ? data
      reg[process + 1] ! data
```
(a)

```
G1: for i in 0 to 6 generate
  process (clk)
  begin
    if clk'event and clk = '1' then
      values(i+1) <= values(i);
    end if;
  end process;
end generate;
```
(b)

```
class shiftreg : public sc_module {
public:
  sc_in<bool> clock;
  sc_in<int> in; sc_out<int> out;

  shiftreg(sc_module_name name, unsigned len)
    : sc_module(name) {
    char buf[10];
    sc_signal<int> * prev = 0;
    for (unsigned int i = 0 ; i < len ; i++) {
      sprintf(buf, "r%d", i);
      reg * regp = new reg(buf);
      regp->clock(clock);
      if (i==0) regp->in(in); else regp->in(*prev);
      if (i < len - 1)
        regp->out(*(prev = new sc_signal<int>));
      else regp->out(out);
} } };
```
(c)

```
void fifo(chan int i, chan int &o, int n) {
  if (n) {
    chan int c;
      for (;;) next c = next i;
    par
      fifo(c, o, n-1);
  } else for (;;) next o = next i;
}
```
(d)

Figure 2. A concurrent FIFO in (a) occam, (b) VHDL, (c) SystemC, and (d) SHIM. Each are fragments.

ing procedure to determine which variables control recursive calls, then we only simulate relevant variables.

Our procedure simplifies program analysis because it produces programs with a static call structure. Hardware implementations demand such programs since true recursion has no real analogue in hardware, where every transistor and wire must be known when the circuit is assembled. Software is less restrictive, but recursive programs are harder to analyze and optimize. For example, because the language we target does not provide a heap, it is possible to statically bound the memory required by a program with a static call graph. Furthermore, knowing the static call structure allows connection information to be computed at compile-time to reduce run-time overhead. Finally, a static call structure, especially when the program is finite-state, greatly simplifies formal verification based on model checking.

We make three contributions: a collection of examples illustrating how to build parallel structures with recursion, a partial-evaluation algorithm that can remove the recursion, and experimental results that show the algorithm is practical.

2. Related Work

Most parallel languages have some facility for elaborating parallel structures. Figure 2 shows code from a variety of languages that each describe more-or-less the same thing: a sequence of one-place buffers connected back-to-back.

Many parallel languages provide a parallel *for* loop that constructs arrays of parallel tasks at run time. Occam's [11] replicated parallel construct (Figure 2a) is typical: it instantiates a set of parallel tasks, passing a unique index to each. Occam insists the base and count expressions (on each side of the *FOR* keyword) be constant and the tasks are only allowed to read the index variable.

Hardware must have its structure defined at compile time, so any algorithmic generation of structure is done then. VHDL [12] provides the *generate* statement, which works as a compile-time *for* loop. The fragment in Figure 2b instantiates seven concurrent processes that each behave like edge-triggered flip-flops. The Verilog 2001 language has a similar feature.

Bjesse et al.'s Lava [4] is a language embedded in Haskell designed for the algorithmic specification of digital circuits. Since Lava is embedded in Haskell, a circuit designer is free to use arbitrary Haskell code to build the structure of the circuit. Furthermore, the generated circuit may be simulated or used to generate VHDL depending on how the library interprets the data structures so generated.

Although Haskell can be used to specify structure in Lava, it cannot be used to specify function, so Lava is purely an elaboration language. Simple Boolean logic gates are the only source of function. By contrast, our work allows SHIM programs to consist of both structure and function.

Li and Leeser's HML [15] is also a functional hardware description language. They included support in the language for generating structure through recursion, much like we do, but their compiler did not support it [14].

Hoe and Arvind's Bluespec [10] is another hardware description language with a powerful elaboration phase. Also based on a Haskell infrastructure, it provides powerful mechanisms for generating structures algorithmically. Unfortunately, few details about its approach have been published.

The SystemC library for hardware simulation [8] introduces a separate elaboration phase between compilation and system execution. SystemC is actually a C++ library; running a simulation consists of compiling and running a SystemC program. When the program starts, it first elaborates the design, building in-memory data structures that represent the structure of the hardware system, then runs an event-based simulation kernel.

Elaboration in SystemC is performed by arbitrary C++ code (usually in constructors), and so is very flexible. Figure 2c shows a *shiftreg* class (a SystemC module) that takes an elaboration-time parameter that controls the number of registers to string together.

For simulation, SystemC's runtime elaboration is convenient, if verbose, but it creates problems for analysis. Moy et al.'s [16] Pinapa tool is one of the few that takes this problem seriously: it extracts the structure of SystemC models by statically analyzing the source code, running the elaboration phase, and later correlating the two. While technically difficult because SystemC was not designed to facilitate this, this is semantically straightforward: all SystemC programs explicitly start the simulation. Pinapa treats all code that runs before this point as elaboration and only looks at its output.

While loop unrolling and procedure inlining are well-known, comparatively little literature addresses unrolling recursion. Most focuses on efficiency. Appel [2] notes that it can reduce stack space requirements and provide a speed-up. Rugina and Rinard [18] focus on improving the speed of divide-and-conquer algorithms by partially inlining recursive calls and fusing conditionals. This leads to more efficient code that spends more time conquering instead of dividing. However, their techniques do not attempt to remove recursion entirely, and so they address a problem different than ours.

Two-level functional languages [17] provide compile- and run-time computation and distinguish them syntactically. While their consideration of combine-time computation resembles ours, we support concurreny and do not require explicit programmer annotations. We suspect our technique is more convenient, but have not performed experiments verifying this.

3. The SHIM Language

The SHIM language [19] provides a cohesive model for both hardware and software systems. Its key strength lies in providing deterministic concurrency: a SHIM program always reacts the same way to the same inputs regardless of scheduling choices; it has no data races.

SHIM has a C-like syntax: a program is composed of functions built from statements and expressions. Neither global variables nor pointers are allowed. Instead, SHIM includes a mechanism for concurrent function calls through the *par* construct and rendezvous-style inter-process communications through the blocking *next* communication operator. Figure 2d shows a SHIM implementation of a parametric FIFO that combines parallelism with recursive function calls. For $n > 0$, it runs a one-place buffer containing an infinite *for* loop ("*for* (;;) *next c* = *next i*") in parallel with a FIFO of size $n-1$, otherwise, it is a single one-place buffer.

The *par* construct starts concurrent tasks. *p par q* starts statements *p* and *q* in parallel, waits for both to complete, then runs the next statement in sequence. These statements may be arbitrary blocks, which may themselves include *par* statements.

To prevent data races, SHIM forbids a variable to be passed by reference to two concurrent tasks. For example,

```
void f(int &x) {}

void g(int x) {}

void main() {
  int x, y;
  f(x); par g(x); par f(y);                        // OK
  f(x); par f(x);          // rejected: x passed by-ref twice
}
```

SHIM adopts an asynchronous concurrency model, à la Kahn networks [13] (SHIM tasks can only block on a single channel), that uses CSP-like rendezvous [9]. Only communication affects the relative execution rates of concurrent tasks. The language does not support shared memory.

A channel resembles a local variable. Passing a channel by value copies its value, which can be modified independently. A channel must be passed by reference to senders.

Communication is blocking: a task that attempts to communicate must wait for all other connected tasks to engage in the communication. If the synchronization completes, the sender's value is broadcast to the receivers.

Like most formalisms with blocking communication, SHIM programs may deadlock. But deadlocks are easier to fix in SHIM because they are deterministic: on the same input, a SHIM program will either always or never deadlock.

Shim's exceptions [20] enable a task to gracefully interrupt its concurrently running siblings. A sibling is "poisoned" by an exception when it attempts to communicate with a task that raised an exception or with a poisoned task. An exception handler runs after all the tasks in its scope have terminated or been poisoned.

While SHIM's communication and exception constructs are novel, they are largely outside the scope of this paper. Our algorithm does not attempt at compile time to track data communicated among tasks, even when it would be possible.

4. Recursive Parallel Structures

We argue concurrency plus recursion provides a powerful mechanism for constructing structures algorithmically and that our elaboration technique makes it possible to use this mechanism in a setting that prohibits recursion. Below, we give examples that use recursion to build such structures.

4.1 Pipelines

Figure 2d shows an n-place buffer using our technique. This can be generalized to n parallel computations. For example, Figure 3 builds a finite pipeline for testing the Collatz conjecture: the "$3n+1$" problem.

The pipeline stage number can select the function of each stage. Figure 4 shows a prime-number sieve that filters according to a list of primes.

Figure 5 is a smarter sieve that uses a pair of processes at each stage to filter the primes from an increasing sequence of numbers. Each filter process ("F") considers the first number it receives as prime since the number made it through all the previous stages, then passes each number that is not divisible by this prime. Each multiplexer process ("M") first passes the newly-identified prime to the output, then passes all those from later stages. The last stage ($n = 0$) returns its (prime) number and throws an exception to terminate the sieve.

4.2 A Sequential/Parallel/Sequential Pipeline

Figure 6 depicts the schedule for a common pattern in concurrent systems: a variant of map-reduce found, for example, in a JPEG decoder. A source process (Huffman decoding in JPEG) divides an input stream into pieces. These pieces are distributed to parallel processor tasks (these perform the IDCT in JPEG) and then reassembled in sequence by a sink process.

Figure 7 shows the SHIM code for a parametric implementation of such a pipeline. The bodies of *src* and *snk* (not shown—they may be arbitrary functions) each update a state variable that is passed by reference. The *src* function throws an exception when it reaches the end of input to terminate the pipeline. The *src_task* and *snk_task* functions are just concurrent wrappers for running sequences of the *src* and *snk* functions. *Src_task* repeatedly takes a state, runs *src*, passes the output to a processor, and passes out its state. *Snk_task* reads the input from the processor, then the state, runs *snk*, and passes the state out.

The *stage* function creates most of the structure. For each stage, it runs *processor* in parallel with a *src_task* and *snk_task*, which generate and consume the input and output for the *processor*. Like earlier examples, it uses a tail-recursive call to build the next stage.

Finally, the main entry point—the *pipeline* function—calls *stage* in parallel with two processes that seed and then feed back the states from the *src* and *snk* processes. Finally, when the *src* function throws *Done* to indicate it has processed all the input, *pipeline* returns the final state.

4.3 A Fast Fourier Transform

The fast Fourier transform is a commonly needed computation that involves complex but highly patterned data flow. Even more so than the largely examples we presented above, the FFT's structure is hierarchical and well-suited to recursion.

We started with a sequential, recursive implementation from Grama et al. [7] (Figure 8) and noted that it first divides the even-

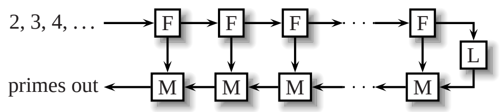
```
int collatz(int i) {
  return i&1 ? 3*i+1 : i/2;
}

void pipeline(chan int i, chan int &o, int n) {
  if (n) {
    chan int c;
      for (;;) next c = collatz(next i);
    par
      pipeline(c, o, n-1);
  } else
    for (;;) next o = next i;
}
```

Figure 3. A pipeline testing the Collatz conjecture

```
void sieve(chan int i, chan int &o, int n,
           int primes[10]) {
  if (n) {
    chan int c;
      for (;;)
        if (next i % primes[n] || i == primes[n])
          next c = i;
    par
      sieve(c, o, n-1, primes);
  } else
    for (;;)
      if (next i % primes[0] || i == primes[0] )
        next o = i;
}
```

Figure 4. A prime-number sieve that takes primes from an array



```
void sieve(chan int num_in, chan int &prime_out,
           int n) throws Done {
  if (n) {
    chan int prime, num_out, prime_in;
    {                                       // Filter process (F)
      next prime = next num_in;
      for (;;)
        if (next num_in % prime)
          next num_out = num_in;
    } par {                            // Multiplexer process (M)
      next prime_out = next prime;
      for (;;) next prime_out = next prime_in;
    } par
        sieve(num_out, prime_in, n-1);
  } else {                                   // Last process (L)
    next prime_out = next num_in;
    throw Done;
  }
}
```

Figure 5. Another prime-number sieve

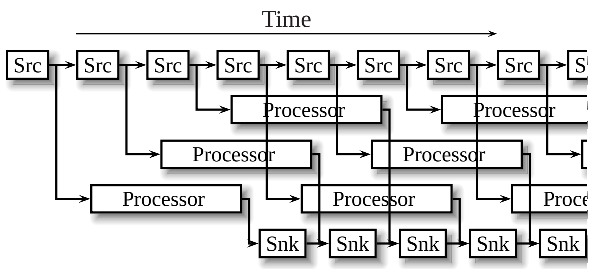


Figure 6. Schedule for a three-processor pipeline

```
int src(src_st &state) throws Done { ... }
int processor(int in) { ... }
void snk(int in, snk_st &state) { ... }

void src_task(chan src_st st_in, chan src_st &st_out,
              chan int &out) throws Done {
  for (;;) {
    next out = src(next st_in);
    next st_out = st_in;
  }
}

void snk_task(chan snk_st st_in, chan snk_st &st_out,
              chan int in) {
  for (;;) {
    snk(next in, next st_in);
    next st_out = st_in;
  }
}

void stage(chan src_st src_in, chan src_st &src_out,
           chan snk_st snk_in, chan snk_st &snk_out,
           uint n) throws Done {
  chan int proc_in, proc_out;
    for (;;) next proc_out = processor(next proc_in);
  par
    if (n) {
      chan src_st src_in1; chan snk_st snk_in1;
        src_task(src_in, src_in1, proc_in);
      par snk_task(snk_in, snk_in1, proc_out);
      par stage(src_in1,src_out,snk_in1,snk_out,n-1);
    } else {
        src_task(src_in, src_out, proc_in);
      par
        snk_task(snk_in, snk_out, proc_out);
    }
}

snk_st pipeline(src_st initial_src, snk_st initial_snk,
                uint n) {
  chan src_st src_in, src_out;
  chan snk_st snk_in, snk_out;
  try {
    next src_in = initial_src;
    for (;;) next src_in = next src_out;
  } par {
    next snk_in = initial_snk;
    for (;;) next snk_in = next snk_out;
  } par {
    stage(src_in, src_out, snk_in, snk_out, n);
  } catch (Done) {}
  return snk_out;
}
```

Figure 7. Code for an *n*-processor pipeline

```
procedure FFT($\langle x_0,\ldots,x_{n-1}\rangle, \langle y_0,\ldots,y_{n-1}\rangle, n, \omega$)
  if $n = 1$ then
    $y_0 = x_0$
  else
    FFT($\langle x_0,x_2,\ldots,x_{n-2}\rangle, \langle q_0,\ldots,q_{n/2}\rangle, n/2, \omega^2$)
    FFT($\langle x_1,x_3,\ldots,x_{n-1}\rangle, \langle t_0,\ldots,t_{n/2}\rangle, n/2, \omega^2$)
    for $i = 0$ to $n-1$ do
      $y_i = q_{(i \bmod (n/2))} + \omega^i \cdot t_{(i \bmod (n/2))}$
```

Figure 8. The sequential FFT from Grama et al. [7] that we parallelized. $\omega = e^{2\pi i/n}$ initially.

and odd-numbered inputs among two sub-procedures, then makes two sequential steps through the results of the sub-procedures because of the "$i/mod(n/2)$" subscripts on q and t.

Figure 9 shows the structure and code we created for the FFT. Each n-point stage instantiates two $n/2$-point FFTs and feeds the result to a "butterfly" processor. Each sample is a complex number, which we represent as two fixed-point numbers (integers) since SHIM does not support floating-point. Each stage expects n samples on its *in* port and produces n samples on *out*.

The demultiplexer task divides the incoming samples into even and odd streams and passes them to two $n/2$-point FFTs. The two output streams are fed to both the butterfly task and a pair of $n/2$-sample buffers. The butterfly task uses these buffers to step through the $n/2$ pairs of samples from the sub-FFTs twice, combining each pair with a with different "twiddle" factor (different complex roots of unity).

This may not be the most efficient FFT algorithm, but it does illustrate that recursion and concurrency are powerful. We used recursion to perform a divide-and-conquer-style decomposition of the FFT, used tail recursion to build the buffers, and took advantage of SHIM's multiway rendezvous facility to make two copies of the results of the sub-procedures.

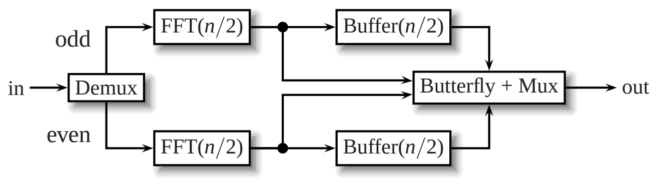
5. Binding-Time Analysis

Our goal is to eliminate recursion while increasing the size of the program as little as possible. Some expansion is usually inevitable since the recursion we target usually amounts to one or more loops, which we unroll.

To limit the amount of expansion we perform during our symbolic simulation process (described in the next section), we perform a binding-time analysis with a slicing procedure [21] to tell us which variables directly or indirectly control recursive calls. By tracking only these variables, we effectively avoid unrolling loops that are irrelevant to recursive calls—the main source of unneeded expansion.

We slice using what is in effect a program dependence graph [6]. Our goal is to determine, for each function, a set of variables that we should track to understand recursion. We compute control dependence by computing dominators using the Lengauer-Tarjan algorithm (we followed the description in Appel [3]) then compute dominance frontiers and control dependence following Cytron et al. [5]. For data dependence, we use the standard dataflow algorithm for reaching definitions taken from Aho et al. [1], although we compute reaching definitions for each statement, not just every basic block.

To identify relevant variables, we first decompose the static call graph of the program into strongly connected components (SCCs) using Tarjan's algorithm. In each SCC, we consider recursive any call site that calls a function within the same SCC. This definition covers both self-recursion and mutually recursive functions.



(a)

```
struct cplx {                            // Fixed−point complex number
  int32 r;
  int32 i;
};

void cplx_buffer(chan cplx in, chan cplx &out, uint n) {
  if (n>1) {
    chan cplx local;
      for (;;) next local = next in;
    par
      cplx_buffer(local, out, n-1);
  } else for (;;) next out = next in;
}

cplx butterfly(cplx x, cplx y, int32 theta) {
  cplx result, w;
  w.r = cos_fixed(theta);
  w.i = sin_fixed(theta);
  result.r = x.r + MUL(w.r, y.r) - MUL(w.i, y.i);
  result.i = x.i + MUL(w.r, y.i) + MUL(w.i, y.r);
  return result;
}

void fft(chan cplx in, chan cplx &out,
         uint n, int32 theta) {
  if (n == 1) {
    for (;;) next out = next in;            // The trivial FFT
  } else {
    chan cplx even, odd, q, t, q1, t1;
      for (;;) {                                  // Demux
        next even = next in;
        next odd = next in;
      }
    par fft(even, q, n/2, theta * 2);        // Even samples
    par fft(odd, t, n/2, theta * 2);          // Odd samples
    par cplx_buffer(q, q1, n/2);            // Buffer for even
    par cplx_buffer(t, t1, n/2);             // Buffer for odd
    par for (;;) {
      for (int i = 0 ; i < n/2 ; i++)        // Butterfly + Mux
        next out = butterfly(next q, next t, theta * i / 2);
      for (int i = n/2 ; i < n ; i++)
        next out = butterfly(next q1, next t1, theta * i / 2);
    }
  }
}
```

(b)

Figure 9. (a) A parallel, recursive restructuring of the FFT from Figure 8, and (b) the corresponding SHIM code.

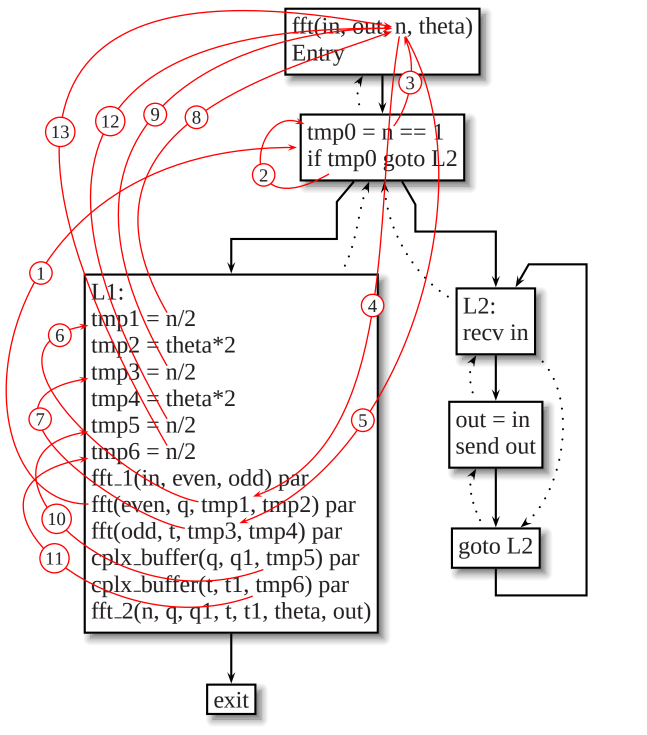


Figure 10. Slicing *fft* in Figure 9b. Solid arrows denote control flow, dotted lines denote control dependence, numbered lines indicate steps in the slicing procedure.

We consider relevant all basic blocks on which these recursive call sites are dependent. Such blocks usually end in conditional control-transfer instructions (e.g., *if* statements); we consider relevant the variables they test.

From these blocks that control recursive calls, we perform the usual fixed-point computation that considers every definition of a relevant variable and any control-transfer instruction on which such definitions are control-dependent.

We slice across procedure boundaries. When we find a dependence on a formal parameter to a function, we mark as relevant the corresponding actual argument passed at every call to that function and continue propagating backwards. Such analysis naturally crosses SCC boundaries.

5.1 Slicing the FFT

To understand our slicing procedure, consider Figure 10, the control-flow graph for the *fft* function from Figure 9.

First, we observe that the calls to *fft* in the L1 block of Figure 10 are recursive (each function is its own SCC and calls itself). This block (L1) is control-dependent on its predecessor—a block that ends with an *if* conditional test (1) (numbers in parentheses refer to the numbered arcs). This tests temporary variable *tmp0*, which is defined in the statement immediately preceeding it (tmp0 = n == 1) (2). On the right of this statement is the variable *n*, which is defined as the third formal argument (3).

Since we reached a formal argument of a function, we propagate dependence from the corresponding actual argument at each of the function's call sites. There are two such sites here, in block L1, which pass temporaries *tmp1* and *tmp3* (4), (5). These are defined in statements in the same block (6), (7), which use *n*, defined as the formal argument (8), (9). Since we have already considered this formal variable, nothing further needs to be done. The definitions for *tmp1* and *tmp3* are also control-dependent on the *if* block, but we also do not need to consider this block again.

```
for (;;) {
  if (next a == 5) c = 1; else c = a;
  c += 5;
  send c;
}
```

(a)

```
L1:                          // nothing live
  recv a
  tmp = a == 5
  ifnot tmp goto else
  c = 1                      // nothing live
  goto endif
else:                        // a live
  c = a
endif:                       // c live
  c = c + 5
  send c
  goto L1
```

(b)

```
L1:
  recv a
  tmp = a == 5
  ifnot tmp goto L2
  c = 6
  send c
  goto L1
L2:
  c = a
  c = c + 5
  send c
  goto L1
```

(c)

Figure 11. (a) A contrived filter process, (b) its representation in our IR with live variables indicated at the start of each basic block, and (c) the IR after applying the simulation procedure with all variables considered relevant. This eliminated one addition and duplicated *send c* but did not duplicate *recv a* because *c* was not live at L1.

Similar reasoning finds a recursive call of *cplx_buffer* in Figure 9. It is also control-dependent on the *if* statement that tests formal argument *n*, so this formal argument is relevant to recursion.

Cplx_buffer is called in L1 of *fft-dependence*. Here, both *tmp5* and *tmp6* are passed as actual arguments for (relevant) actual argument *n*. These are defined above (10), (11), which in turn both reference *n* (12), (13).

For function *fft*, our procedure finds variables *tmp0*, *tmp1*, *tmp3*, *tmp5*, *tmp6*, and *n* are relevant to removing recursion. The static elaborator, described below, only considers these variables when partially evaluating the program.

6. Our Static Elaborator

The binding-time analyzer described in the last section finds a set of "relevant variables" for each function by slicing the program from every recursive call site. Using this information, our static elaborator simulates the basic blocks in each function, tracking the exact value of each relevant variable and creating a constant-propagated copy of each block along the way. Literals and variables whose values are discovered during this procedure are treated as constants used to simplify statements during the procedure; all other values, including those communicated through channels, the contents of structures and arrays, and the values of all irrelevant variables, are treated as unknown. Code that manipulates unknown values is copied into the generated code.

Once a particular basic block has been duplicated as many times as its limit, no further copies are made (we currently use one global user-specified number for all basic blocks, but we could use block-specific limits). After that, if our algorithm discovers control being passed to the block with different known variable values, instead passes control to a copy of the block that assumes all live variables take unknown values, which amounts to a copy of the original block.

Our algorithm operates on our intermediate representation (IR)—a fairly standard three-address code. As usual, each block

```
procedure simulate(func, block, value)
  if block $(func, block, value)$ does not exist then
    Increase $copies(func, block, value)$
    let maxed-out = true if there are over limit copies
    let successors = set of successor blocks for block
    for each sequential statement stmt in block do
      match statement stmt with
      $d = s$ :                                                    Move
        if $value(s) \neq \bot$ and $d$ is relevant then
          let $value(d) = value(s)$                          value known
        else
          let $value(d) = \bot$
          add "$simp(d) = simp(s)$"                        compute value
      $d = s_1 \odot s_2$ :                                 Binary operator
        if $value(s_1) \neq \bot$, $value(s_2) \neq \bot$, and $d$ relevant then
          let $value(d) = value(s_1) \odot value(s_2)$
        else
          let $value(d) = \bot$
          add "$simp(d) = simp(s_1) \odot simp(s_1)$"
      default :                                       All other statements
        add "stmt"                                             just copy
    if maxed-out then
      for each variable $v$ s.t., $value(v) \neq \bot$ do
        add "$v = value(v)$"
    match control-transfer statement of block with
    Goto $l$ :                                                      Goto
      add "Goto $l$"
    if $s$ goto $l$ :                                        Conditional
      if $value(s) = \bot$ then                        predicate unknown
        add "if $s$ goto $l$"
      else if $value(s) \neq 0$ then                predicate known true
        add "Goto $l$" and let successors = { block for $l$ }
      else                                         predicate known false
        let successors = { fall-through block }
    Recv $d$ :                                                   Receive
      let $value(d) = \bot$ and add "Recv $d$"             value unknown
    Send $s$ :                                                      Send
      if $value(s) \neq \bot$ then                           value known
        add "$s = value(s)$"
      add "Send $s$"
    $f_1(a_1^1, \ldots, a_j^1)$ par $\cdots$ par $f_n(a_1^n, \ldots, a_k^n)$ :      Call
      for each pass-by-ref $a$ with $value(a) \neq \bot$ do
        add "$a = value(a)$"                      if an exception thrown
      for each function call $f(a_1, \ldots, a_k)$ do
        for each variable $v$ live in $f$'s entry block do
          let $a_i$ be the actual arg. for formal arg. $v$
          let $value'(v) = \begin{cases} \bot & \text{if } maxed\text{-}out, \\ value(a_i) & \text{otherwise.} \end{cases}$
        add $(f, 0, value')$ to call-successors
        create $f'(b_1, \ldots, b_k)$, where
          $b_k = \begin{cases} simp(a_k) & \text{if } a_k \text{ is pass-by-value,} \\ a_k & \text{otherwise.} \end{cases}$
      add "$f_1'(b_1^1, \ldots, b_j^1)$ par $\cdots$ par $f_n'(b_1^n, \ldots, b_k^n)$"
      let $value(a) = \bot$ for each pass-by-ref arg. $a$
    for each successor block $b$ do
      let $value'(v) = value(v)$ for each live var. $v$ in $b$
      add $(func, b, value')$ to to-visit
    add call-successors signatures to to-visit & entries
```

Figure 12. The basic block simulation procedure

```
procedure remove-recursion(functions, limit)
  divide functions into basic blocks
  compute control dependence and reaching definitions
  slice from every recursive call to find relevant variables
  compute live variables for each basic block
  for each live variable $v$ in the entry block of main do
    let $value(v) = \bot$
  let to-visit = $\{(main, 0, value)\}$
  while to-visit is not empty do
    remove a signature $(func, block, value)$ from to-visit
    call simulate$(func, block, value)$
  for each entry point $e$ in entries do
    create a new function for $e$ from the simulation results
  Remove unreferenced labels
  Inline functions that consist only of calls
  return all the newly-created functions
```

Figure 13. The recursion removal procedure

starts with an optional label, its body consists of sequential statements, and it may end with a control-transfer instruction: an *if*, a *switch*, a *goto*, or a parallel function call.

To further reduce code expansion, we only track relevant variables that are live. In particular, if a relevant variable is no longer live after a block, we forget its value, which generally allows us to merge blocks that would otherwise have a different signature (i.e., the value of each relevant variable).

Figure 11 shows the effect of considering live variables. Assuming all variables are relevant when we simulate the code fragment in Figure 11a, we find control can reach the *endif* label with either $c = 1$ or c unknown (since a is received and therefore unknown). We thus make two copies of this block, one for each case, and propagate this information to discover $c = 6$ or c is unknown just before the *send*. We still know c's value in one case when the simulation passes control to *L1*, but because c is not live there, we treat these two cases equally and produce only one copy for the basic block at *L1*.

Figure 12 depicts the basic block simulation procedure in all its glory.[1] This first checks whether the requested block has already been computed. If not, it determines whether we have reached the limit and should not create any more variant children (the *maxed-out* flag).

Each sequential statement that assigns to a relevant variable is simulated and the results stored if all operands are known, otherwise the variable is marked as unknown.

If this block is to generate no more children, code is inserted that "spills" the values of each known variable in the form of a load-constant statement so the runtime can effectively take over the computation.

Control-transfer instructions are more interesting. The rule for *if* determines if the predicate is known at simulation time and replaces it with a *goto* if possible. Being able to do this is critical for stopping the unrolling process. The rule for *recv* marks its variable as unknown; we do not attempt to simulate data passed through communication channels.

The rule for parallel function calls is complicated because pass-by-reference variables must be "spilled" before the function is called because if the variable is relevant, the function could throw an exception and lose the simulated value. Any known values of pass-by-value parameters are passed. Finally, the values of every pass-by-reference argument are marked as unknown—our simulation does not track values returned by functions.

[1] Almost: we omit similar rules for unary operators, multiway branches, and *ifnot* statements.

```
fifo(chan int i, chan int &o, int n)
chan int c
local int tmp
    ifnot n goto L1
    tmp = n - 1
    fifo_a(i, c) par fifo(c, o, tmp)
    goto L2
L1: recv i
    o = i
    send o
    goto L1
L2:

fifo_a(chan int i, chan int &c)
L1: recv i
    c = i
    send c
    goto L1
```

(a)

```
fifo(chan int i, chan int &o, int n)
chan int c
    fifo_a(i, c) par fifo_2(c, o, 2)

fifo_2(chan int i, chan int &o, int n)
chan int c
    fifo_a(i, c) par fifo_1(c, o, 1)

fifo_1(chan int i, chan int &o, int n)
chan int c
    fifo_a(i, c) par fifo_0(c, o, 0)

fifo_0(chan int i, chan int &o, int n)
L1: recv i
    o = i
    send o
    goto L1

fifo_a(chan int i, chan int &c)
L1: recv i
    c = i
    send c
    goto L1
```

(b)

```
fifo(chan int i, chan int &o, int n)
chan int c
chan int c1
chan int c2
    fifo_a(i, c) par fifo_a(c, c1) par
    fifo_a(c1, c2) par fifo_0(c2, o, 0)

fifo_0(chan int i, chan int &o, int n)
L1: recv i
    o = i
    send o
    goto L1

fifo_a(chan int i, chan int &c)
L1: recv i
    c = i
    send c
    goto L1
```

(c)

Figure 14. The IR at various stages in removing recursion from the FIFO example: Figure 2d. (a) Initial dismantling into a recursive procedure produces two functions. *fifo_a* is code for the first branch of the *par*. (b) After symbolic simulation and expansion, the values of *n* and *tmp* have been established throughout, resulting in four versions of the original *fifo* procedure, one for each value of *n*. (c) After function inlining, three of the *fifo* functions have been collapsed into one.

Finally, the values of variables for each successor are taken from the simulation. Only the values of live variables are preserved to help reduce the size of the generated code.

6.1 Assembling the Pieces

Figure 13 shows our overall recursion removal procedure. It first divides every function into basic blocks (send and receive operations are considered control-transfer instructions) then computes relevant variables by building the program dependence graph and slicing with respect to all recursive function calls (Section 5).

After computing live variables, the simulation procedure begins by starting the program from the entry point with all live variables set to $\perp$. The simulation procedure uses the usual worklist algorithm (here, *to-visit* is the set of all function signatures remaining to be processed) in which the *simulate* procedure (Figure 12) is called until the *to-visit* set is empty.

Once the simulation procedure completes, each function is reconstructed by assembling the blocks produced during the simulation procedure according to their control dependencies. Finally, unreferenced labels (arising from reassembling the control-flow graph) are removed call-only functions are inlined.

Parallel recursive functions are usually dismantled into multiple functions: one that just handles the recursive calls and the updates of variables that control the recursion and others that perform the actual work. The *fft* function in Figure 9 is typical: With the exception of the copy in-to-out base case, the function just calls itself recursively. All the "heavy lifting" is done in the non-recursive *fft_1* and *fft_2* functions, which perform the demultiplexing and butterfly-multiplexing operations, respectively.

As a result, recursive functions are often transformed into just calls to other functions. While it would not be incorrect to leave them as-is, a function that only calls another can be beneficially inlined. The final step in our algorithm performs such inlining, which noticably reduces the number of functions in the final program.

6.2 Derecursing the FIFO

Figure 14 shows our IR at various points in the recursion removal procedure. Figure 14a is the IR immediately after being dismantled from the code in Figure 2d. Our compiler dismantles *par* blocks into multiple functions so every function remains sequential. Here, *fifo_a* handles the "for (;;) next c = next i;" statement just before the *par* Figure 2d.

Slicing identified variables *n* and *tmp* as relevant and our static elaborator tracks their values to produce the code in Figure 14b. Here, the original *fifo* function has been transformed into four different copies, one for each value of *n* encountered during symbolic simulation ($n = 3, 2, 1, 0$). The first three copies boiled down to nothing more than the parallel function call; the fourth is just the code at label L1.

It is not uncommon for many functions to simply call other functions, so we implemented a simple function inlining procedure that merges calls to functions that do nothing but call other functions in parallel. Figure 14c shows the result of this merging. Here, the first three copies of *fifo* have been merged into one that calls four functions in parallel: three instances of *fifo_a* and one of *fifo_0*, the base case.

7. Experimental Results

To evaluate our procedure, we implemented it in Objective CAML and applied it our examples. Table 1 shows the results. We compared the size of the IR before removing recursion (the "source" columns), with the results of naïve expansion with the limit set to thirty-two (i.e., all variables considered relevant—the "expanded" columns), the results of following relevant variables (the "sliced" columns), and the results of applying function inlining to that (the "inlined" columns).

The "fns." columns list the number of functions after various stages. These counts are large because they include functions for the runtime system, the standard I/O procedures, and helper input

Example	Fig.	*n*	Source			Expanded (32)		Sliced			Inlined		Runtime
			Fns.	Sts.	Vars.	Fns.	Sts.	Fns.	Sts.	Rel.	Fns.	Sts.	
Pipeline	2d	3	14	87	30	17	401	17	86	2	11	84	40ms
Collatz	3	10	14	88	37	24	523	24	96	2	14	93	60
Sieve1	4	9	13	117	42	30	679	30	312	2	21	310	110
Sieve2	5	15	14	83	35	29	528	29	94	2	17	92	80
SeriesPar	7	3	20	104	65	26	400	26	106	3	14	110	60
FFT	9	16	20	255	148	113	1948	34	356	9	26	353	170

n = unrolling factor   Fns. = number of functions   Sts. = number of IR statements
Vars. = total number of arguments + local variables   Rel. = number of relevant variables after slicing

Table 1. Experimental Results

and output functions. Furthermore, only parallel function calls introduce parallelism in our compiler. Internally, we dismantle functions with *par* constructs into functions that call other functions in parallel. All these extra functions are counted to make the numbers realistic.

The "sts." columns list the total number of IR statements across all functions in the program—a rough measure of program and executable size. IR statements are three-address, involving at most a single arithmetic operation. Parallel call is the most complex statement that is still counted as a single one. It may call an arbitrary number of functions and pass an arbitrary number of arguments, although each argument must be a (temporary) variable or constant.

The "vars." column lists the total number of variables (formals, locals, and temporaries) across all functions. The "rel." column lists the number of variables the slicing procedure reported as relevant to recursive calls. For these examples, the slicer is able to select relevant variables very judiciously, keeping the expansion of the program under control.

The running time for our specialization algorithm seems negligible as part of a compilation flow. The "runtime" column lists the time it took the compiler to parse and dismantle the source program, run our procedure, and dump the resulting IR. These times are for running the OCAML bytecode interpreter on a 2.5 GHz Pentium 4 machine.

These results suggest only tracking relevant variables (i.e., heeding the results of slicing) greatly reduces the size of the generated code. Tracking relevant variables resulted in over an 80% reduction in code size on the largest example (FFT); smaller examples were reduced more modestly.

The effect of inlining can substantially reduce the number of functions, suggesting it is also worthwhile. Sieve1 had the largest reduction: 9 of the 30 functions disappeared after inlining. By design, this restricted form of inlining does not have much effect on overall program size.

The "*n*" column lists the argument passed to the main recursive procedure in each example. It is roughly the unrolling factor: how many concurrent copies of the core operation are made. The number of statements after inlining, surprisingly, does not correlate much because *n* only makes copies of part of the program and our elaboration process removes code that controls the expansion.

Overall, these results suggest our procedure is practical. While the increase in code size can be non-negligible (nearly a factor of three for Sieve1), it remains surprisingly modest, even for fairly large expansion factors. Consider Sieve2: it expands into a chain of fifteen process pairs, yet the program size remains modest (83 vs. 92) because much of the program is simulated away and many functions remain called from many places.

8. Conclusions

We presented an algorithm for removing recursion from concurrent programs written in the SHIM programming language. Our goal was to enable analysis tools such as formal verifiers, resource analyzers, and hardware synthesizers that demand non-recursive programs to also handle a certain class of recursive programs.

We presented a number of examples (templates, really) that illustrate how recursion is a powerful way of generating parallel structures algorithmically. While such structures could be generated by macro-style instantiation, we believe using existing language mechanisms is more elegant and easier for the programmer since it does not demand learning a new, often crippled, sublanguage.

Our algorithm works in two phases: first, we use a slicing procedure to determine which variables participate in the control of the recursive procedure and whose values should be bound at compile time. Second, we partially evaluate the program using symbolic simulation to track the relevant variables determined in the first phase and generate an aggressively constant-propagated version of the program from the simulation results.

Experimental results on small examples suggest our procedure is practical. The increase in code size from unrolling the recursion stays modest, even for fairly large expansion factors (e.g., ten), and the runtimes for these small examples are negligible. As a result, it should be practical to employ this procedure as part of the normal compilation process.

Next steps include more carefully tracking the contents of arrays and structures and better understanding communication patterns. While the tail recursive style of writing these expanding programs is powerful, some programmers may find it awkward. We plan to explore alternate syntax for describing structures such as pipelines.

References

[1] Alfred V. Aho, Monica Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts, second edition, 2006.

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

[3] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in Haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pages 174–184, Baltimore, Maryland, 1998.

[5] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[7] Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar. *Introduction to Parallel Computing*. Addison-Wesley, second edition, 2003.

[8] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer, Boston, Massachusetts, 2002.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978.

[10] James C. Hoe and Arvind. Hardware synthesis from term rewriting systems. In *VLSI '99: Proceedings of the IFIP TC10/WG10.5 Tenth International Conference on Very Large Scale Integration*, pages 595–619, Deventer, The Netherlands, 2000. Kluwer, B.V.

[11] INMOS Limited. *occam 2 Reference Manual*. Prentice Hall, 1988.

[12] The Institute of Electrical and Electronics Engineers (IEEE), 345 East 47th Street, New York, New York. *IEEE Standard VHDL Reference Manual (1076–1987)*, 1988.

[13] Gilles Kahn. The semantics of a simple language for parallel programming. In *Information Processing 74: Proceedings of IFIP Congress 74*, pages 471–475, Stockholm, Sweden, August 1974. North-Holland.

[14] Yanbing Li. HML: An innovative hardware design language and its translation to VHDL. Master's thesis, Cornell University, August 1995.

[15] Yanbing Li and Miriam Lesser. HML: An innovative hardware design language and its translation to VHDL. In *Proceedings of the International Conference on Computer Hardware Description Languages and their Applications (CHDL)*, Chiba, Japan, August 1995.

[16] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. Pinapa: An extraction tool for SystemC descriptions of systems-on-a-chip. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 317–324, Jersey City, New Jersey, September 2005.

[17] Flemming Nielson and Hanne Riis Rielson. *Two-Level Functional Languages*. Cambridge University Press, 1992.

[18] Radu Rugina and Martin Rinard. Recursion unrolling for divide and conquer programs. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 2017 of *Lecture Notes in Computer Science*, pages 34–48, Yorktown Heights, New York, August 2000.

[19] Olivier Tardieu and Stephen A. Edwards. R-SHIM: Deterministic concurrency with recursion and shared variables. In *Proceedings of the 4th International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, page 202, Napa, California, July 2006.

[20] Olivier Tardieu and Stephen A. Edwards. Scheduling-independent threads and exceptions in SHIM. In *Proceedings of the International Conference on Embedded Software (Emsoft)*, pages 142–151, Seoul, Korea, October 2006.

[21] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.