# Hardware Synthesizable Exceptions using Continuations

Paul Teng
McGill University, Canada

Christophe Dubach
McGill University / MILA, Canada

McGill

# Code that is potentially erroneous

```
int mydiv(int r) {
  return 100 / r;
}


int main(int n) {
  return mydiv(n);
}
```

# Code that is potentially erroneous

```
int mydiv(int r) {
  return 100 / r;
}
```

```
int main(int n) {
  return mydiv(n);
}
```

Maybe your code…

- Divides by zero
- Overflows during computation
- Accesses invalid parts of an array
- Something else?

# Solution: Runtime Exceptions

```
int mydiv(int r) {
  if (r == 0)
    throw std::runtime_error("Cannot divide by zero");
  return 100 / r;
}

int main(int n) {
  return mydiv(n);
}
```

**throw**: an **abort** that can be handled later

```
int mydiv(int r) {
  if (r == 0)
    throw std::runtime_error("Cannot divide by zero");
  return 100 / r;
}

int main(int n) {
  return mydiv(n);
}
```

```
int mydiv(int r) {
  if (r == 0)
    throw std::runtime_error("Cannot divide by zero");
  return 100 / r;
}
```

**try / catch**: the **handler** for the **aborts**

```
int main(int n) {
  try { return mydiv(n); }
  catch (...) { return INT_MAX; }
}
```

# Runtime exceptions: try / catch / throw

- Either **throws** on error or **continues** on success (error and success cases are **disjoint**)

Typical software implementation depends a **call stack**

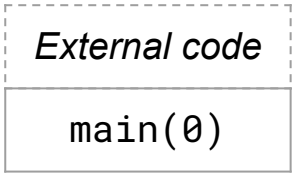
+ HLS prefers plain **state machines** over **call stack**

= **HLS tools do not support exceptions**

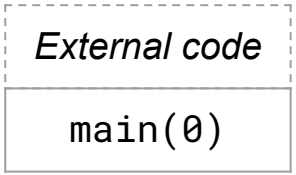
# Stepping through runtime exceptions

```
1. int mydiv(int r) {
2.   if (r == 0)
3.     throw std::runtime_error("Cannot divide by zero");
4.   return 100 / r;
5. }

6. int main(int n) {
7.   try {
8.     return mydiv(n);
9.   } catch (...) {
10.    return INT_MAX;
11.  }
12. }
```

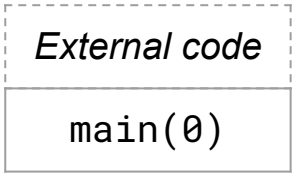
```
 1. int mydiv(int r) {
 2.   if (r == 0)
 3.     throw std::runtime_error("Cannot divide by zero");
 4.   return 100 / r;
 5. }

 6. int main(int n) {
 7.   try {
 8.     return mydiv(n);
 9.   } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```



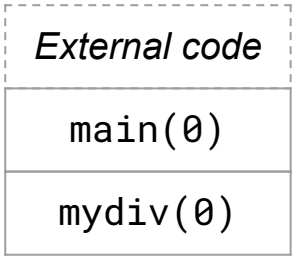
```
 1. int mydiv(int r) {
 2.   if (r == 0)
 3.     throw std::runtime_error("Cannot divide by zero");
 4.   return 100 / r;
 5. }

 6. int main(int n) {
 7.   try {
 8.     return mydiv(n);
 9.   } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```



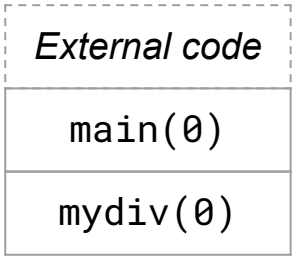
```
 1. int mydiv(int r) {
 2.   if (r == 0)
 3.     throw std::runtime_error("Cannot divide by zero");
 4.   return 100 / r;
 5. }

 6. int main(int n) {
 7.   try {
 8.     return mydiv(n);
 9.   } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```



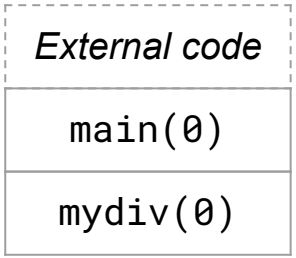
```
 1. int mydiv(int r) {
 2.   if (r == 0)
 3.     throw std::runtime_error("Cannot divide by zero");
 4.   return 100 / r;
 5. }

 6. int main(int n) {
 7.   try {
 8.     return mydiv(n);
 9.   } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```



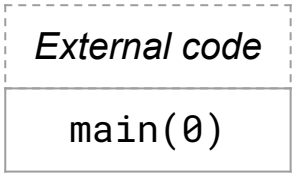
```
 1. int mydiv(int r) {
 2.   if (r == 0)
 3.     throw std::runtime_error("Cannot divide by zero");
 4.   return 100 / r;
 5. }

 6. int main(int n) {
 7.   try {
 8.     return mydiv(n);
 9.   } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```



```
 1. int mydiv(int r) {
 2.   if (r == 0)
 3.     throw std::runtime_error("Cannot divide by zero");
 4.   return 100 / r;
 5. }

 6. int main(int n) {
 7.   try {
 8.     return mydiv(n);
 9.   } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```

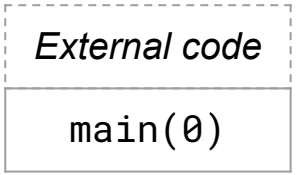


```
 1. int mydiv(int r) {
 2.   if (r == 0)
 3.     throw std::runtime_error("Cannot divide by zero");
 4.   return 100 / r;
 5. }

 6. int main(int n) {
 7.   try {
 8.     return mydiv(n);
 9.   } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```

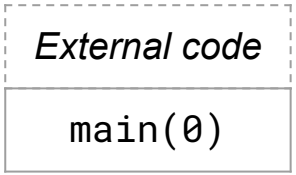



Here, we **unwind** the call stack

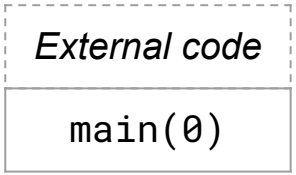
```
 1. int mydiv(int r) {
 2.   if (r == 0)
 3.     throw std::runtime_error("Cannot divide by zero");
 4.   return 100 / r;
 5. }

 6. int main(int n) {
 7.   try {
 8.     return mydiv(n);
 9.   } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```



```
1.  int mydiv(int r) {
2.    if (r == 0)
3.      throw std::runtime_error("Cannot divide by zero");
4.    return 100 / r;
5.  }

6.  int main(int n) {
7.    try {
8.      return mydiv(n);
9.    } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```



```
 1. int mydiv(int r) {
 2.   if (r == 0)
 3.     throw std::runtime_error("Cannot divide by zero");
 4.   return 100 / r;
 5. }

 6. int main(int n) {
 7.   try {
 8.     return mydiv(n);
 9.   } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```



```
1.  int mydiv(int r) {
2.    if (r == 0)
3.      throw std::runtime_error("Cannot divide by zero");
4.    return 100 / r;
5.  }

6.  int main(int n) {
7.    try {
8.      return mydiv(n);
9.    } catch (...) {
10.     return INT_MAX;
11.   }
12. }
```

*External code*

How to avoid the call stack
when runtime exceptions do **unwinding**?

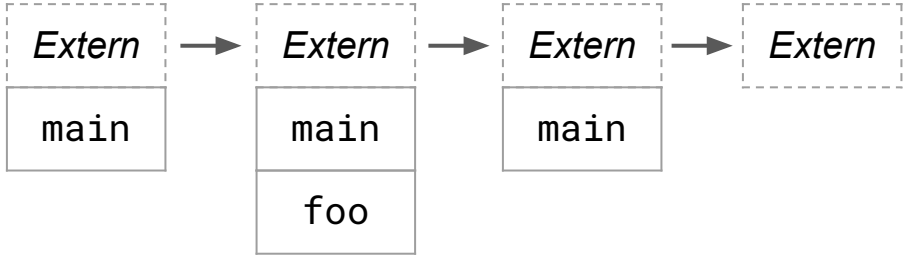
# Main Idea: Use CPS

1. What is CPS?
2. How to add exceptions to CPS?

# Continuation Passing Style

turns the program into a state machine

```
fun foo(x: Int): Int =
  x + 1

fun main(n: Int): Int =
  foo(n) + 2
```




Original program
(**not in CPS**)

# Automatic conversion into CPS

```
fun foo(x: Int): Int =
  x + 1
```

```
cont foo(x: Int,
         cb: cont(Int)) =
  goto cb(x + 1)
```

```
fun main(n: Int): Int =
  foo(n) + 2
```

```
cont main(n: Int,
          k: cont(Int)) =
  cont j(result: Int) =
    goto k(result + 2)
  goto foo(n, j)
```

# Automatic conversion into CPS

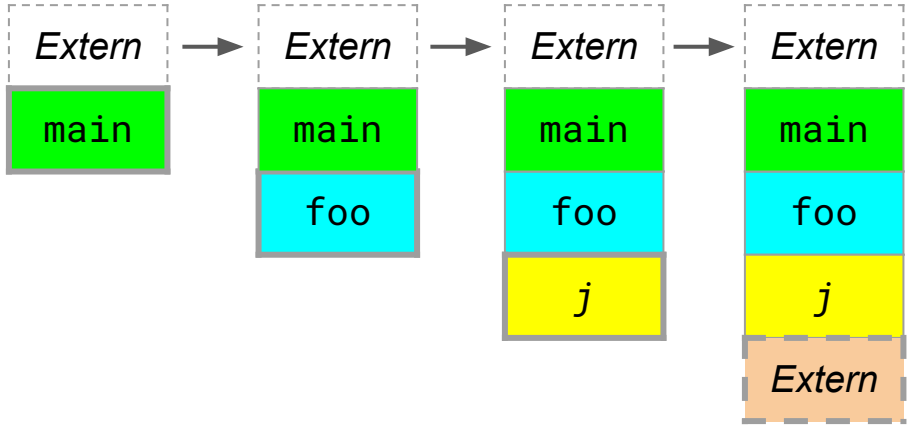
```
fun foo(x: Int): Int =
  x + 1
```

```
cont foo(x: Int,
         cb: cont(Int)) =
  goto cb(x + 1)
```

```
fun main(n: Int): Int =
  foo(n) + 2
```

```
cont main(n: Int,
          k: cont(Int)) =
  cont j(result: Int) =
    goto k(result + 2)
  goto foo(n, j)
```

```
cont foo(x: Int,
         cb: cont(Int)) =
  goto cb(x + 1)

cont main(n: Int,
          k: cont(Int)) =
  cont j(result: Int) =
    goto k(result + 2)
  goto foo(n, j)
```

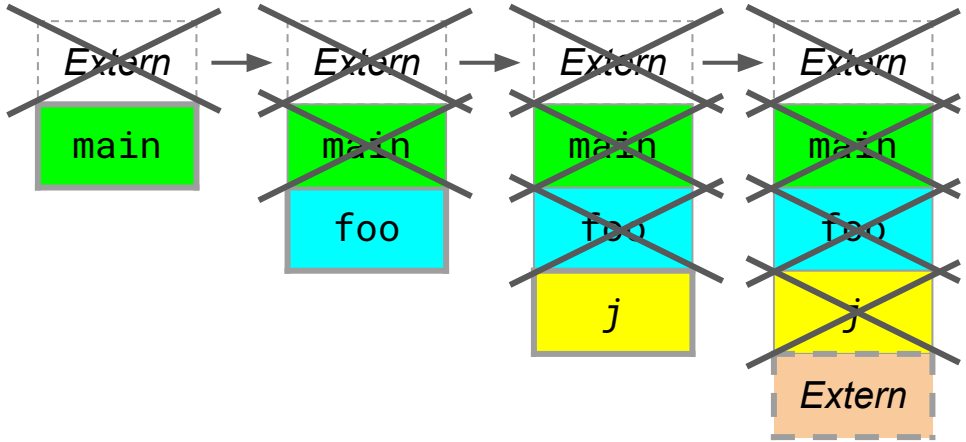



CPS uses more stack frames?

Continuation Passing Style

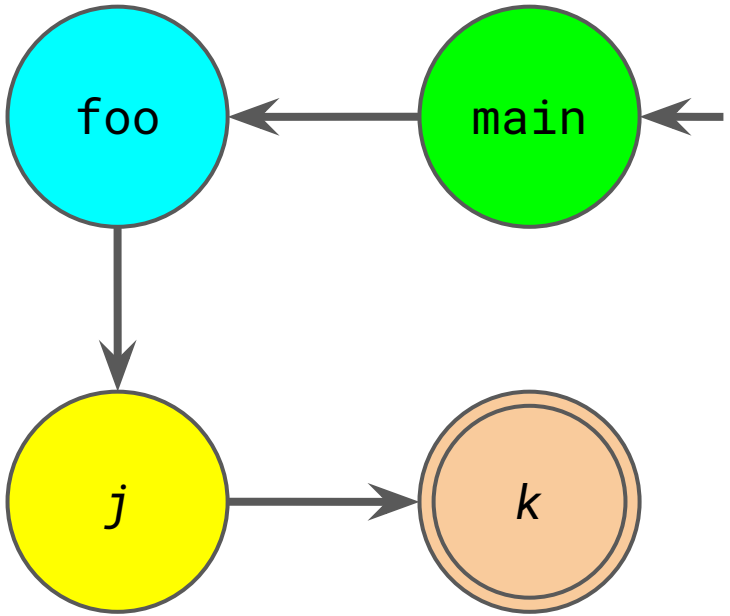
```
cont foo(x: Int,
         cb: cont(Int)) =
  goto cb(x + 1)

cont main(n: Int,
          k: cont(Int)) =
  cont j(result: Int) =
    goto k(result + 2)
  goto foo(n, j)
```

Continuation Passing Style

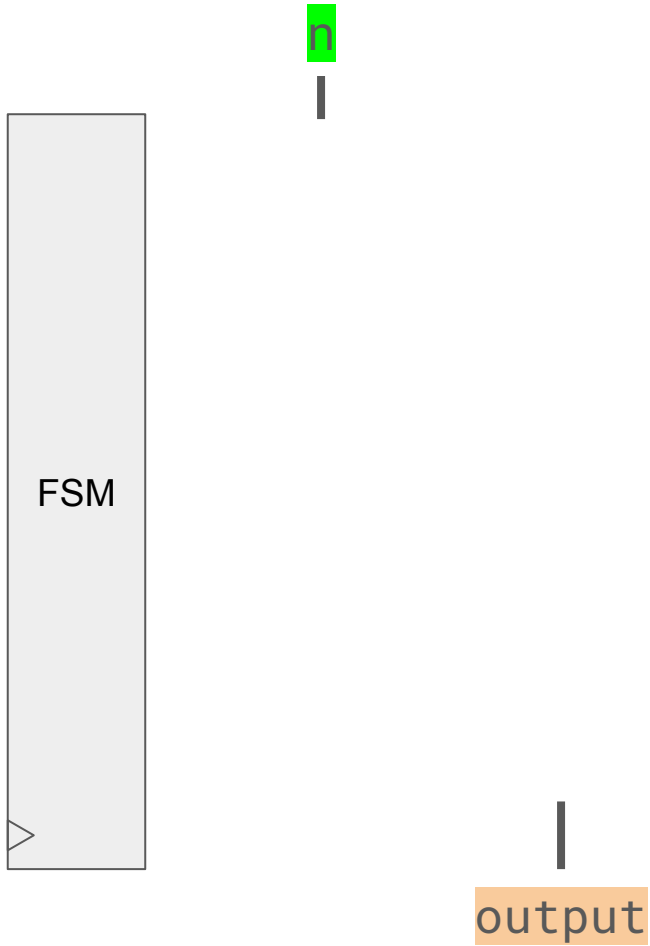



Crossed-out frames are useless!
You only need **one** frame!

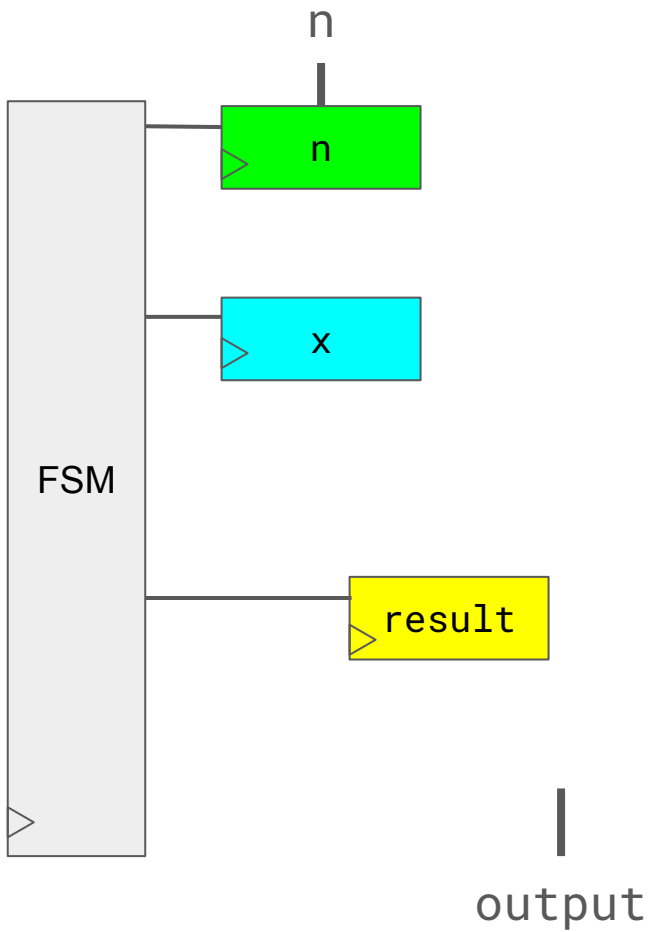
```
cont foo(x: Int,
         cb: cont(Int)) =
  goto cb(x + 1)

cont main(n: Int,
          k: cont(Int)) =
  cont j(result: Int) =
    goto k(result + 2)
  goto foo(n, j)
```



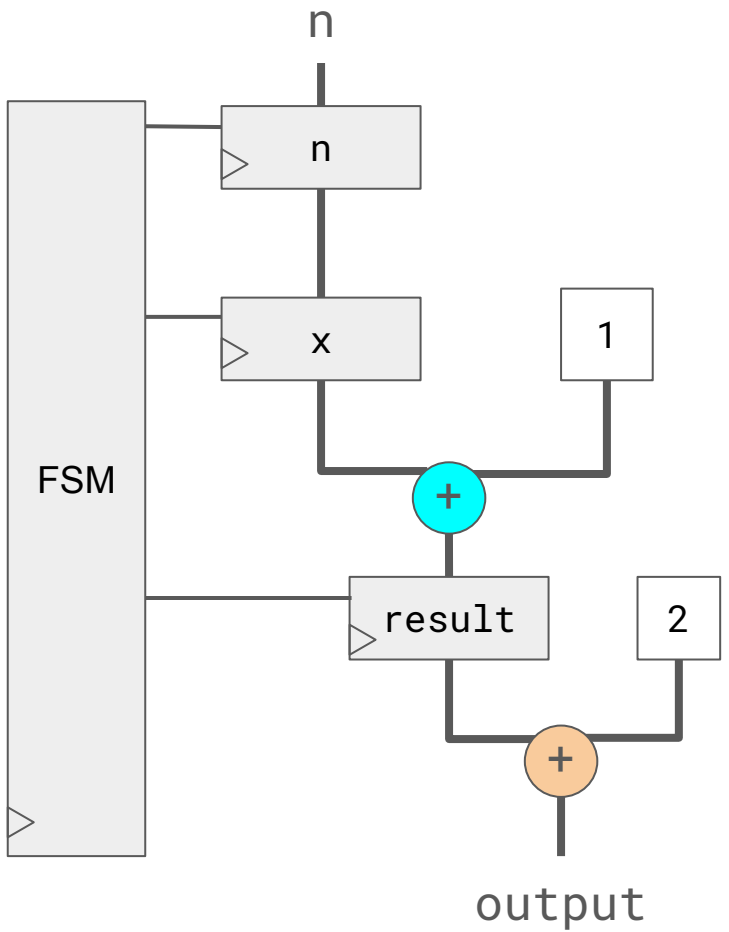
```
cont foo(x: Int,
         cb: cont(Int)) =
  goto cb(x + 1)

cont main(n: Int,
          k: cont(Int)) =
  cont j(result: Int) =
    goto k(result + 2)
  goto foo(n, j)
```



```
cont foo(x: Int,
         cb: cont(Int)) =
  goto cb(x + 1)

cont main(n: Int,
          k: cont(Int)) =
  cont j(result: Int) =
    goto k(result + 2)
  goto foo(n, j)
```



```
cont foo(x: Int,
         cb: cont(Int)) =
  goto cb(x + 1)

cont main(n: Int,
          k: cont(Int)) =
  cont j(result: Int) =
    goto k(result + 2)
  goto foo(n, j)
```



# Add exceptions to CPS

by adding more continuations…

```
fun mydiv(r: Int): Int =
  if r == 0 then
    throw DivByZeroException
  else
    100 / r
```

Original program

```
cont mydiv(r: Int,
           k: cont(Int)) =
  if r == 0 then
    ???
  else
    goto k(100 / r)
```

CPS

```
fun mydiv(r: Int): Int =

  if r == 0 then
    throw DivByZeroException
  else
    100 / r
```

Original program

```
cont mydiv(r: Int,
           h: cont(Exn),
           k: cont(Int)) =
  if r == 0 then
    goto h(DivByZeroException)
  else
    goto k(100 / r)
```

CPS

```
fun main(n: Int): Int =

  try mydiv(n)
  catch
    DivByZeroException ->
      INT_MAX
```

Original program

```
cont main(n: Int,
          h: cont(Exn),
          k: cont(Int)) =
  cont handler(ex: Exn) =
    match ex {
      case DivByZeroException ->
        goto k(INT_MAX)
      case _ -> goto h(ex)
    }
  goto mydiv(n, handler, k)
```

CPS

# No more try / catch / throw

do the rest in terms of continuations

# Compiler optimization: inlining

```
cont main(n: Int,
          h: cont(Exn),
          k: cont(Int)) =
  cont handler(ex: Exn) =
    match ex {
      case DivByZeroException ->
        goto k(INT_MAX)
      case _ -> goto h(ex)
    }
  goto mydiv(n, handler, k)
```

```
cont mydiv(r: Int,
           h: cont(Exn),
           k: cont(Int)) =
  if r == 0 then
    goto h(DivByZeroException)
  else
    goto k(100 / r)
```

```
cont main(n: Int,
          h: cont(Exn),
          k: cont(Int)) =
  cont handler(ex: Exn) =
    match ex {
      case DivByZeroException ->
        goto k(INT_MAX)
      case _ -> goto h(ex)
    }
if n == 0 then
  goto handler(DivByZeroException)
else
  goto k(100 / n)
```

# Compiler optimization: inlining

# Compiler optimization: inlining

```
cont main(n: Int,
          h: cont(Exn),
          k: cont(Int)) =
  cont handler(ex: Exn) =
    match ex {
      case DivByZeroException ->
        goto k(INT_MAX)
      case _ -> goto h(ex)
    }
  if n == 0 then
    goto handler(DivByZeroException)
  else
    goto k(100 / n)
```

# Compiler optimization: inlining

```
cont main(n: Int,
          h: cont(Exn),
          k: cont(Int)) =
  if n == 0 then
    match DivByZeroException {
      case DivByZeroException ->
        goto k(INT_MAX)
      case _ -> goto h(DivByZeroException)
    }
  else
    goto k(100 / n)
```

# Compiler optimization: partial evaluation

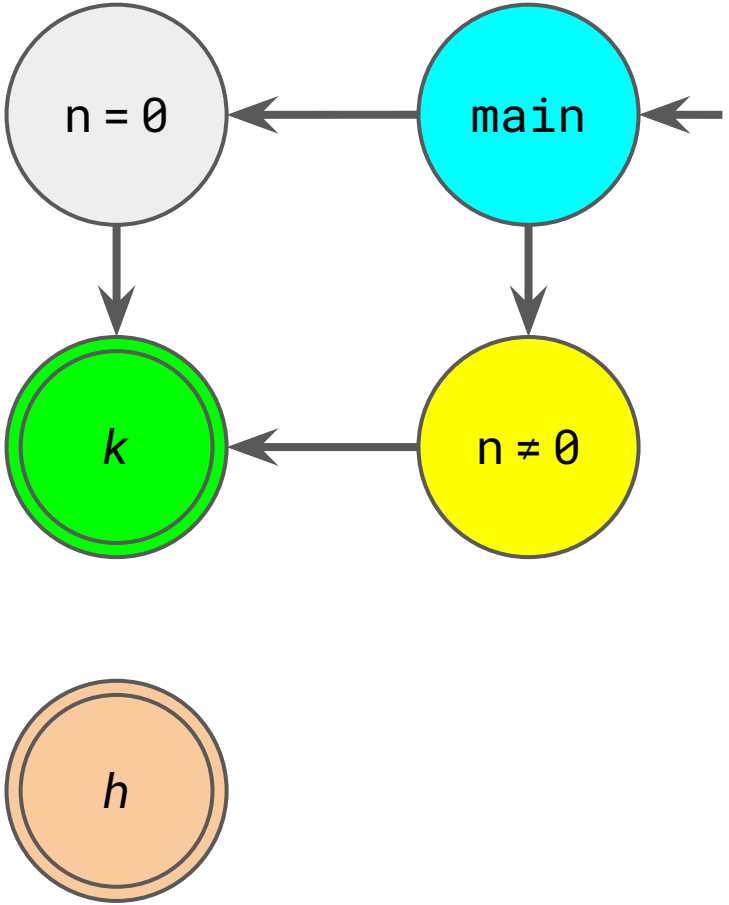
```
cont main(n: Int,
          h: cont(Exn),
          k: cont(Int)) =
  if n == 0 then
    match DivByZeroException {
      case DivByZeroException ->
        goto k(INT_MAX)
      case _ -> goto h(DivByZeroException)
    }
  else
    goto k(100 / n)
```

```
cont main(n: Int,
          h: cont(Exn),
          k: cont(Int)) =
  if n == 0 then
    goto k(INT_MAX)
  else
    goto k(100 / n)
```

# Compiler optimization: partial evaluation

Typical compiler optimizations **removed** the use of exceptions **without** being exception-centric

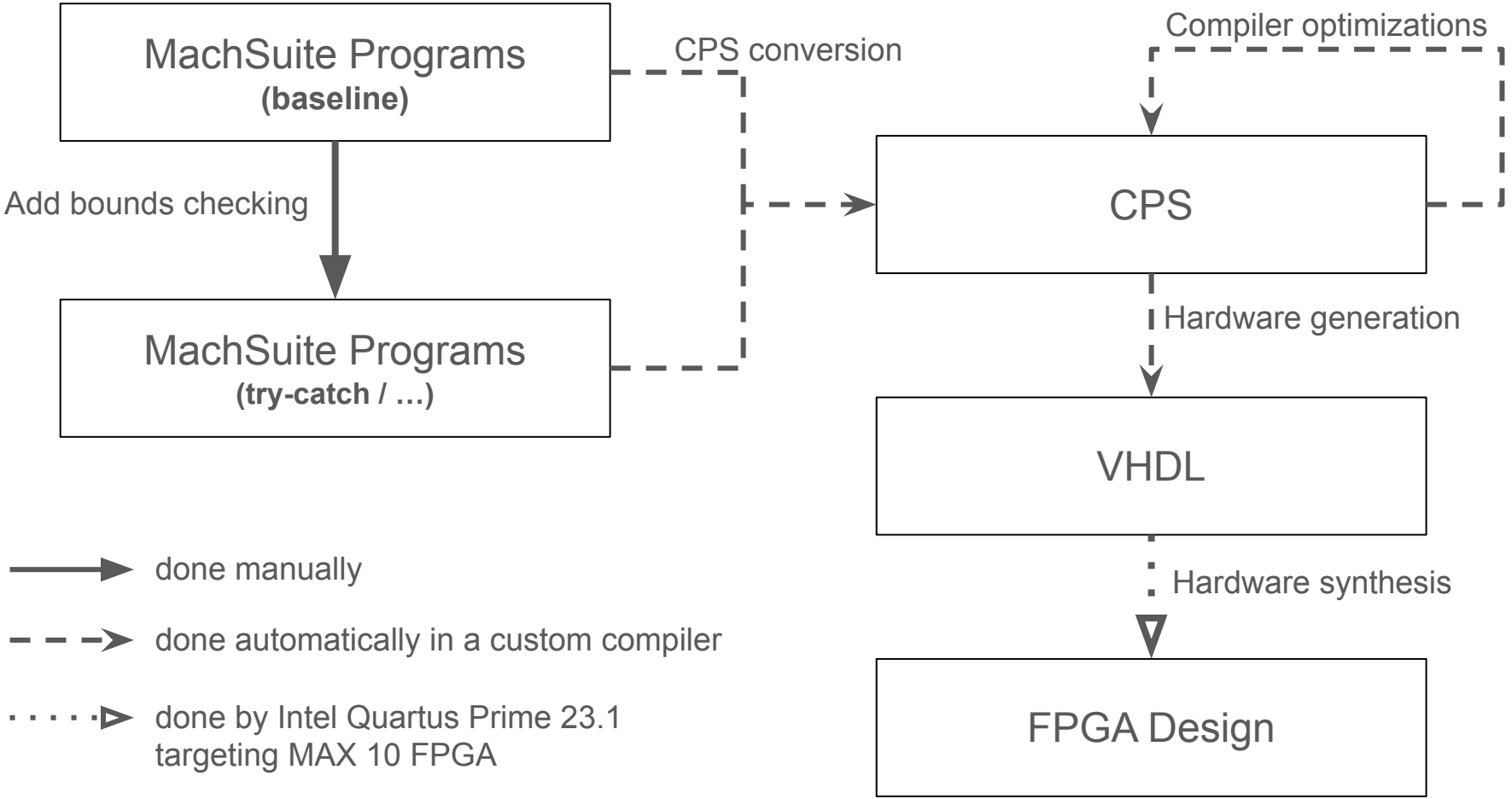
```
cont main(n: Int,
          h: cont(Exn),
          k: cont(Int)) =
  if n == 0 then
    goto k(INT_MAX)
  else
    goto k(100 / n)
```

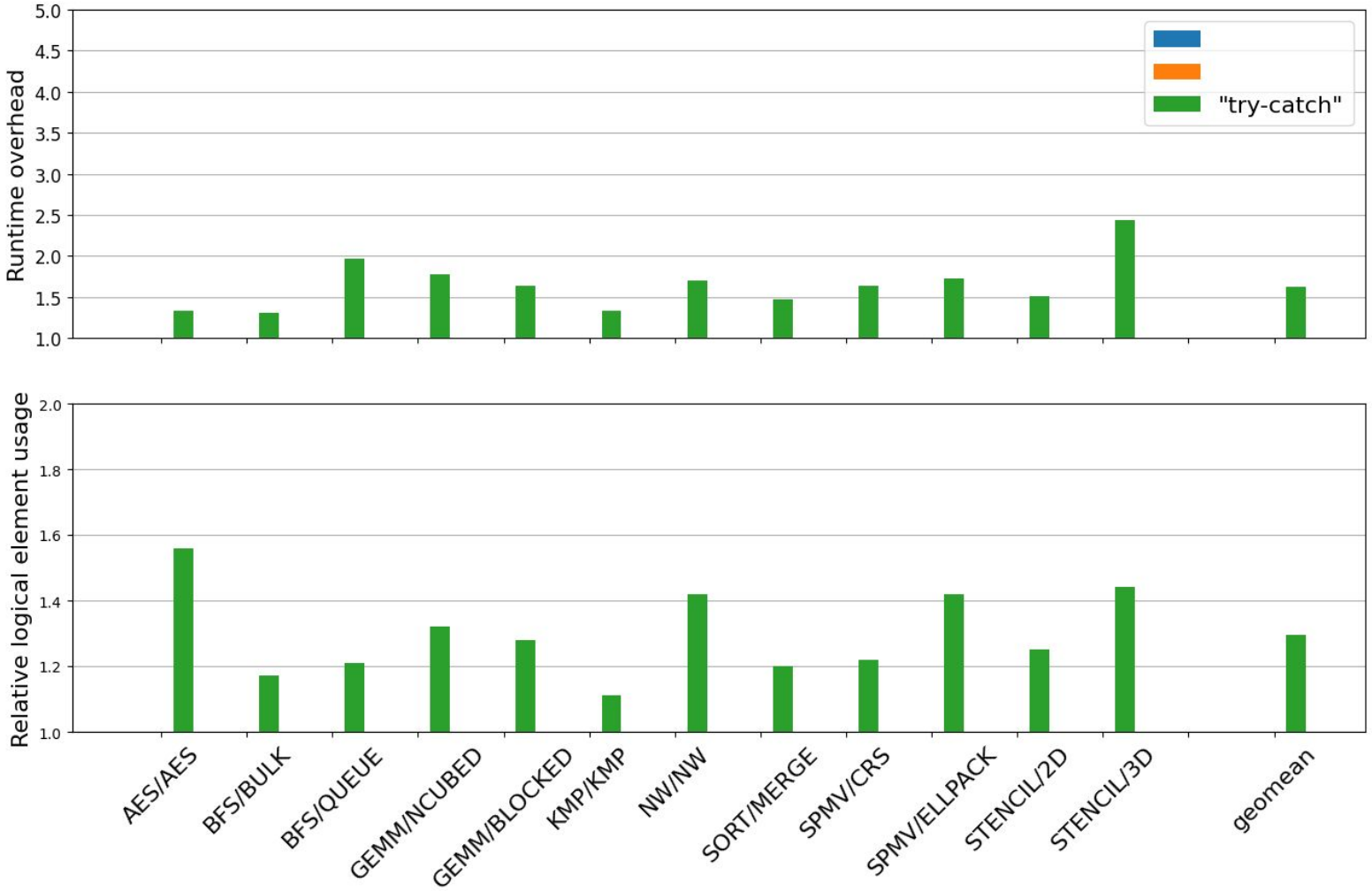


# MachSuite

- “Benchmark suite intended for accelerator-centric research”
- Total of 19 programs written in C
- None of which trigger erroneous cases

- Manually ported 12 of the 19 programs → **no** error handling (baseline)
- Add bounds checking as source of error → **with** error handling (try-catch / …)


MachSuite Programs
(baseline)
Add bounds checking
MachSuite Programs
(try-catch / …)
CPS conversion
Compiler optimizations
CPS
Hardware generation
VHDL
Hardware synthesis
FPGA Design
done manually
done automatically in a custom compiler
done by Intel Quartus Prime 23.1
targeting MAX 10 FPGA


Runtime overhead
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
"try-catch"
Relative logical element usage
2.0
1.8
1.6
1.4
1.2
1.0
AES/AES
BFS/BULK
BFS/QUEUE
GEMM/NCUBED
GEMM/BLOCKED
KMP/KMP
NW/NW
SORT/MERGE
SPMV/CRS
SPMV/ELLPACK
STENCIL/2D
STENCIL/3D
geomean

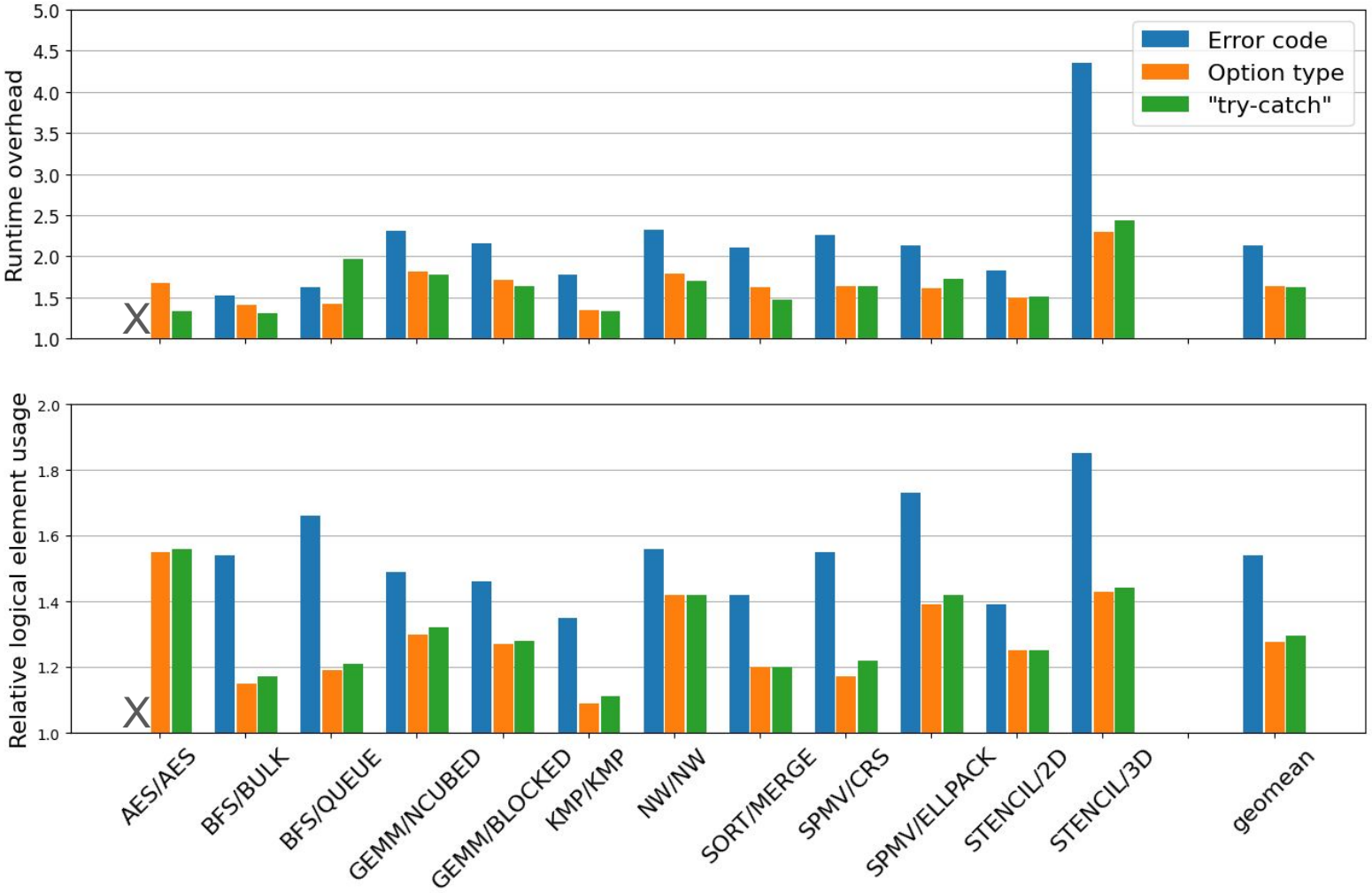
# Error code? Option type?

```
Int MYDIV_ERR;

fun mydiv(r: Int): Int =
  if r == 0
  then { MYDIV_ERR = 1; 0 }
  else { MYDIV_ERR = 0; 100 / r }


fun main(n: Int): Int = {
  Int quot = mydiv(n);
  if MYDIV_ERR == 1 then INT_MAX
                    else quot
}
```

Error code

```
fun mydiv(r: Int): Option(Int) =
  if r == 0 then None
            else Some(100 / r)


fun main(n: Int): Int =
  match mydiv(n) {
    case Some(quot) -> quot
    case None -> INT_MAX
  }
```

Option type


Runtime overhead
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
X
Error code
Option type
"try-catch"
Relative logical element usage
2.0
1.8
1.6
1.4
1.2
1.0
X
AES/AES
BFS/BULK
BFS/QUEUE
GEMM/NCUBED
GEMM/BLOCKED
KMP/KMP
NW/NW
SORT/MERGE
SPMV/CRS
SPMV/ELLPACK
STENCIL/2D
STENCIL/3D
geomean

# Conclusion

1. You write code **without** continuations
2. Compiler turns **`return`** into *k* continuations
3. Compiler turns **`try`** / **`catch`** / **`throw`** into *h* continuations
4. Compiler does optimization, hardware generation, …
   in terms of continuations

Source code of the custom compiler on GitHub:
plankp/Hardware-Synthesizable-Exceptions-using-Continuations