Compiling for Reconfigurable Computing: A Survey

JOÃO M. P. CARDOSO

Universidade do Porto

PEDRO C. DINIZ

Instituto Superior Técnico and INESC-ID

and

MARKUS WEINHARDT

Fachhochschule Osnabrück

Reconfigurable computing platforms offer the promise of substantially accelerating computations through the concurrent nature of hardware structures and the ability of these architectures for hardware customization. Effectively programming such reconfigurable architectures, however, is an extremely cumbersome and error-prone process, as it requires programmers to assume the role of hardware designers while mastering hardware description languages, thus limiting the acceptance and dissemination of this promising technology. To address this problem, researchers have developed numerous approaches at both the programming languages as well as the compilation levels, to offer high-level programming abstractions that would allow programmers to easily map applications to reconfigurable architectures. This survey describes the major research efforts on compilation techniques for reconfigurable computing architectures. The survey focuses on efforts that map computations written in imperative programming languages to reconfigurable architectures and identifies the main compilation and synthesis techniques used in this mapping.

This research was partially supported by FCT, the Portuguese Science Foundation, under research grants PTDC/EEA-ELC/71556/2006, PTDC/EEA-ELC/70272/2006, and POSI/CHS/48018/2002.

This work was performed while J. Cardoso was with Universidade do Algarve and later with IST/UTL.

Authors' addresses: J. M. P. Cardoso, Dep. de Eng. Informática, Faculdade de Engenharia, Universidade do Porto, Rua Dr. Roberto Frias, 4200-465 Porto, Portugal; email: jmpc@acm.org; P. C. Diniz, INESC-ID, Rua Alves Redol n. 9, 1000-029 Lisboa, Portugal; email: pedro.diniz@ist.utl.pt; M. Weinhardt Fak. IuI, Postfach 1940, 49009 Osnabrück, Germany; email: mweinhardt@computer.org.

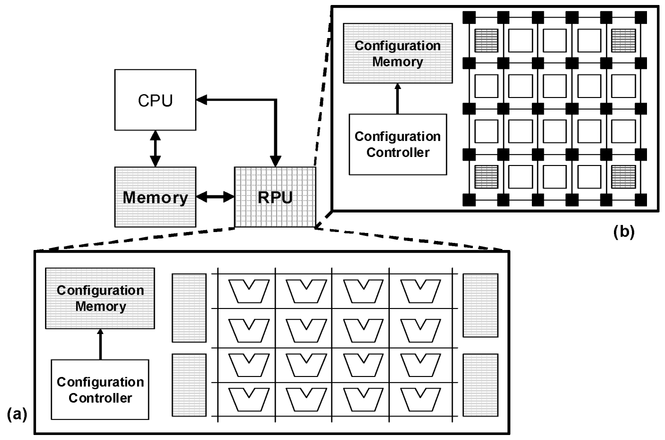
DOI 10.1145/1749603.1749604 http://doi.acm.org/10.1145/1749603.1749604

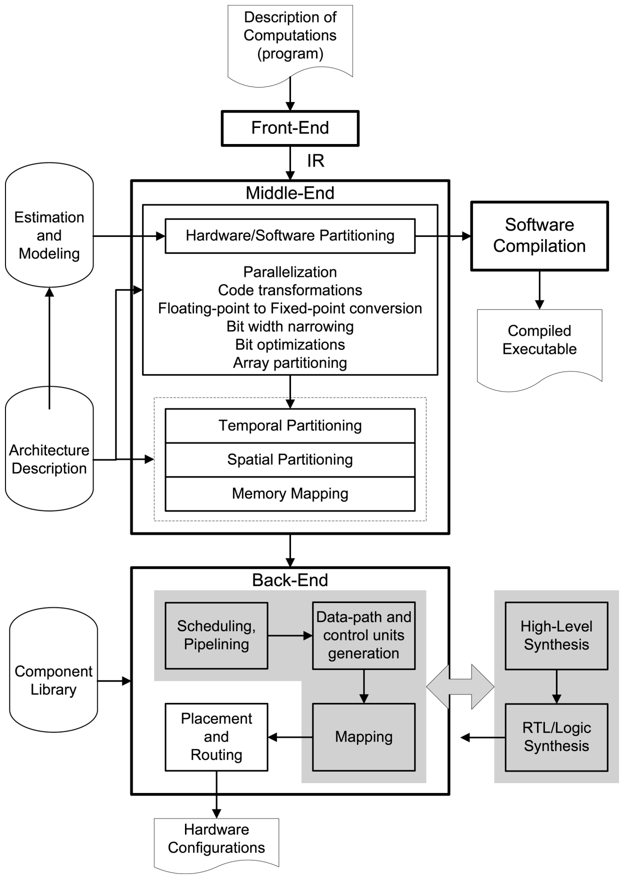
The main characteristic of reconfigurable computing platforms [Lysaght and Rosenstiel 2005; Gokhale and Graham 2005] is the presence of reconfigurable hardware (*reconfigware*), that is, hardware that can be reconfigured on-the-fly (i.e., dynamically and on-demand) to implement specific hardware structures. For example, a given signal processing application might require only 12-bit fixed-point precision arithmetic and use custom rounding modes [Shirazi et al. 1995], whereas other application codes might make intensive use of a 14-bit butterfly routing network used for fast parallel computation of a fast Fourier transform (FFT). In either case, a traditional processor does not have direct support for these structures, forcing programmers to use hand-coded routines to implement the basic operations. In the case of an application that requires a specific interconnection network, programmers must encode in a procedure the sequential evaluation of the data flow through the network. In all these examples, it can be highly desirable to develop dedicated hardware structures that can implement specific computations for performance, but also for other metrics such as energy.

During the past decade a large number of reconfigurable computing systems have been developed by the research community, which demonstrated the capability of achieving high performance for a selected set of applications [DeHon 2000; Hartenstein 2001; Hauck 1998]. Such systems combine microprocessors and *reconfigware* in order to take advantage of the strengths of both. An example of such hybrid architectures is the Napa 1000 where a RISC (reduced instruction set computer) processor is coupled with a programmable array in a coprocessor computing scheme [Gokhale and Stone 1998]. Other researchers have developed reconfigurable architectures based solely on commercially available field-programmable-gate-arrays (FPGAs) [Diniz et al. 2001] in which the FPGAs act as processing nodes of a large multiprocessor machine. Approaches using FPGAs are also able to accommodate on-chip microprocessors as *softcores* or *hardcores*. In yet another effort, researchers have developed dedicated reconfigurable architectures using as internal building blocks multiple functional units (FUs) such as adders and multipliers interconnected via programmable routing resources (e.g., the RaPiD architecture [Ebeling et al. 1995], or the XPP reconfigurable array [Baumgarte et al. 2003; XPP]).

Overall, reconfigurable computing architectures provide the capability for spatial, parallel, and specialized computation, and hence can outperform common computing systems in many applications. This type of computing effectively holds extensive parallelism and multiple flows of control, and can leverage the existent parallelism at several levels (operation, basic block, loop, function, etc.). Most of the existing reconfigurable computing systems have multiple on- and off-chip memories, with different sizes and accesses schemes, and some of them with parameterized facilities (e.g., data width). *Reconfigware* compilers can leverage the synergies of the reconfigurable architectures by exploting multiple flows of control, fine- and coarse-grained parallelism, customization, and by addressing the memory bandwidth bottleneck by caching and distributing data.

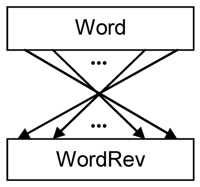
Programming *reconfigware* is a very challenging task, as current approaches do not rely on the familiar sequential programming paradigm. Instead, programmers must


CPU
Memory
RPU
Configuration Memory
Configuration Controller
(a)
Configuration Memory
Configuration Controller
(b)


Description of Computations (program)
Front-End
IR
Middle-End
Estimation and Modeling
Hardware/Software Partitioning
Software Compilation
Compiled Executable
Parallelization
Code transformations
Floating-point to Fixed-point conversion
Bit width narrowing
Bit optimizations
Array partitioning
Architecture Description
Temporal Partitioning
Spatial Partitioning
Memory Mapping
Back-End
Component Library
Scheduling, Pipelining
Data-path and control units generation
High-Level Synthesis
Placement and Routing
Mapping
RTL/Logic Synthesis
Hardware Configurations

```
int Reverse(int Word) {
      int WordRev = 0;
            for(int i=0; i<32; i++) {
                  WordRev |= (Word & 1);
                  WordRev << 1;
                  Word >> 1;
            }
      return WordRev;
}
```

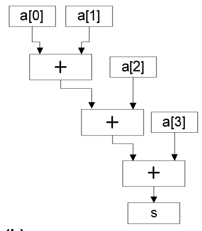
(a)



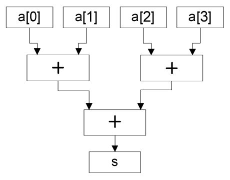
(b)

s = a[0] + a[1] + a[2] + a[3]

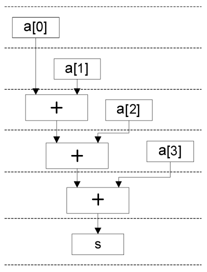
(a)



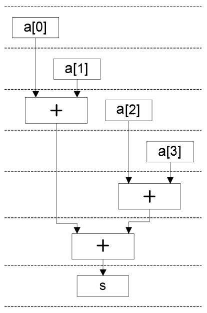
(b)



(c)



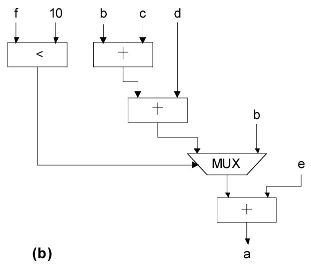
(d)



(e)

```
...
if(f<10)
  a = b + c + d;
else
  a = b;
a = a + e;
...
```

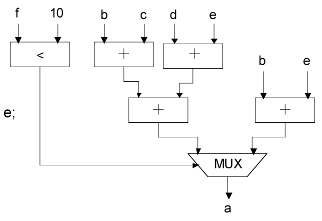
(a)



(b)

```
...
if(f<10)
  a = b + c + d + e;
else
  a = b + e;
...
```

(c)



(d)

```
#define W 3
...
int a[W][W], b[W][W], c[W];
...
for(x=0; x < W; x++) {
      sum = 0;
      for(y=0; y < W; y++) {
            sum += (a[x][y] * b[y][x]);
      }
      c[x] = sum;
}
```
(a)

```
#define W 3
...
int a[W][W], b[W][W], c[W];
...
for(x=0; x < W; x++) {
      sum   = (a[x][0] * b[0][x]);
      sum += (a[x][1] * b[1][x]);
      sum += (a[x][2] * b[2][x]);
      c[x] = sum;
}
```
(b)

```
…
int img[N][N];
…
for(j=0; j < N; j++) {
    …
    for(i=0; i < N; i++) {
        … = img[j][i];
    }
    …
}
```

(a)

```
…
int imgOdd[N][N/2], imgEven[N][N/2];
…
for(j=0; j < N; j++) {
    …
    for(i=0; i < N; i+=2) {
        … = imgOdd[j][i/2];
        … = imgEven[j][i/2];
    }
    …
}
```

(b)

```
…
int img[N][N];
…
for(j=0; j < N; j++) {
    …
    for(i=0; i < N; i++) {
        …= img[j][i];
    }
    …
}
```

(a)

```
…
int imgA[N][N], imgB[N][N];
…
for(j=0; j < N; j++) {
    …
    for(i=0; i < N; i+=2) {
        … = imgA[j][i];
        … = imgB[j][i+1];
    }
    …
}
```

(b)

```
…
int vec[N][4], k[4];
…
for(j=0; j < N; j++) {
    …
    for(i=0; i < 4; i++) {
        … = vec[j][i] * k[i];
    }
    …
}
```

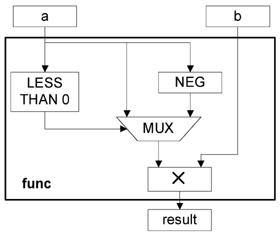
(a)

```
…
int vec[N][4], k[4], k0, k1, k2, k3;
…
k0 = k[0]; k1 = k[1]; k2 = k[2]; k3 = k[3];
for(j=0; j < N; j++) {
    …
    … = vec[j][0] * k0;
    … = vec[j][1] * k1;
    … = vec[j][2] * k2;
    … = vec[j][3] * k3;
    …
}
```

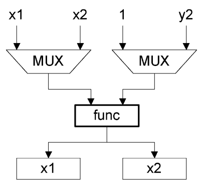
(b)

```
...
int func(int a, int b) {
  if (a < 0)
    return a * b;
  else
    return (-a) * b;
}
...
x1 = func(x1, 1);
...
x2 = func(x2, y2);
...
```

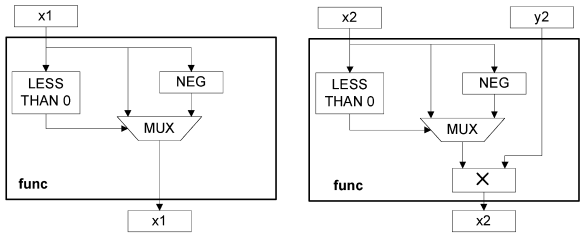
(a)



(b)



(c)



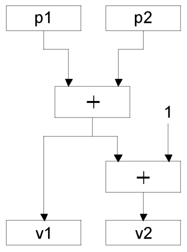
(d)

```
...
x = a + b;
y = x + 1;
...
z  = k + t;
w = z + 1;
...
```

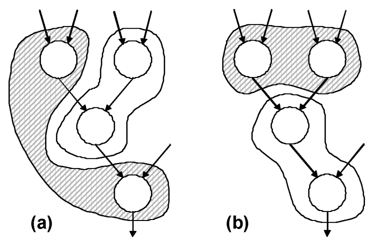
(a)

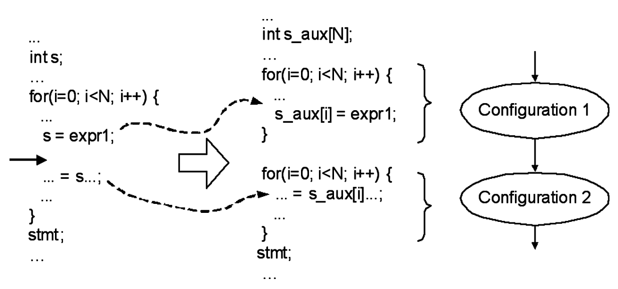
```
...
void func(int *v1, int *v2,
             int p1, int p2) {
   int temp;
   temp = p1 + p2;
   *v1 = temp;
   *v2 = temp + 1;
}
...
func(&x, &y, a, b,);
...
func(&z, &w, k, t);
...
```

(b)



(c)

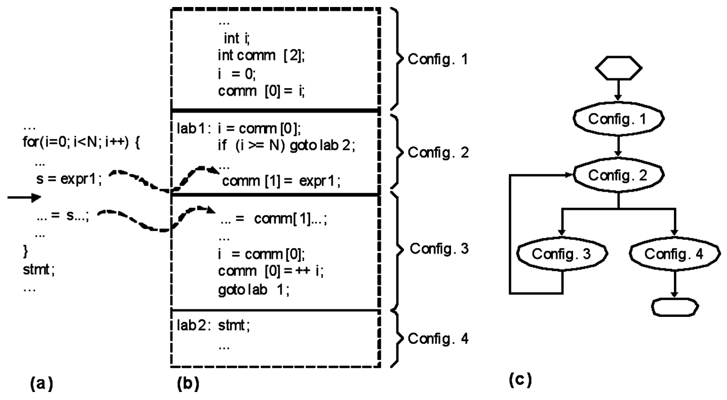

(a)
(b)



```
...
int s;
...
for(i=0; i<N; i++) {
  ...
  s = expr1;

  ... = s...;
  ...
}
stmt;
...
```

```
...
int s_aux[N];
...
for(i=0; i<N; i++) {
  ...
  s_aux[i] = expr1;
}
for(i=0; i<N; i++) {
  ... = s_aux[i]...;
  ...
}
stmt;
...
```

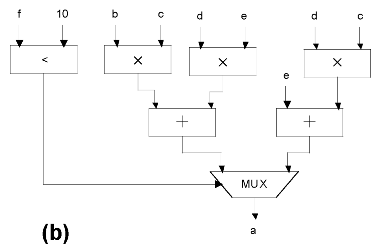


```
...
for(i=0; i<N; i++) {
   ...
   s = expr1;
   ... = s...;
   ...
}
stmt;
...
```

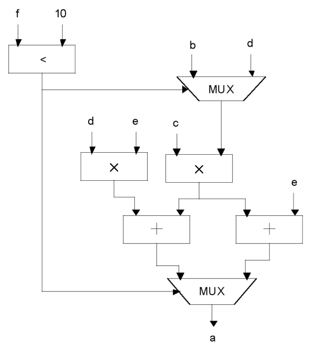
```
         ...
          int i;
         int comm [2];
         i  = 0;
         comm [0] = i;
lab1:    i = comm [0];
         if (i >= N) goto lab 2;
         ...
          comm [1] = expr1;
          ... = comm[1]...;
         ...
         i = comm [0];
         comm [0] = ++ i;
         goto lab 1;
lab2:    stmt;
         ...
```

```
...
if(f<10)
  a  = b * c + d * e;
else
  a  = d * c + e;
...
```

(a)



(b)



(c)

```
...
for(int i=0; i<N; i++) {
      C[i] = A[i]*B[i];
}
...
```

(a)

```
...
int tmp1 = A[0]; // prologue
int tmp2 = B[0]; // prologue
for(int i=1; i<N; i++) {
      C[i-1] = tmp1*tmp2;
      tmp1 = A[i];
      tmp2 = B[i];
}
C[N-1] = tmp1*tmp2; // epilogue
...
```

(b)

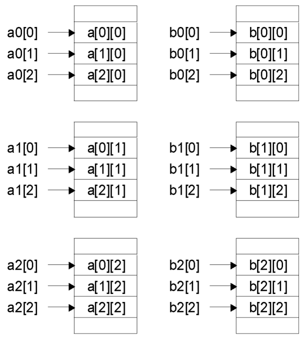
					,

```
#define W 3
…
int a[W][W], b[W][W], c[W];
…
for(x=0; x < W; x++) {
    sum   = (a[x][0] * b[0][x]);
    sum += (a[x][1] * b[1][x]);
    sum += (a[x][2] * b[2][x]);
    c[x] = sum;
}
```

(a)

```
#define W 3
…
int a0[W], b0[W];
int a1[W], b1[W];
int a2[W], b2[W];
int c[W];
…
for(x=0; x < W; x++) {
   sum   = (a0[x] * b0[x]);
   sum += (a1[x] * b1[x]);
   sum += (a2[x] * b2[x]);
   c[x] = sum;
}
```

(b)



(c)

;

```
int:5 a[N]; // 5 bit integer
if((N % 6) == 0) {
    for(i=0; i < N; i += 6) {
        ... = a[i+0];
        ... = a[i+1];
        ... = a[i+2];
        ... = a[i+3];
        ... = a[i+4];
        ... = a[i+5];
    }
}
```

(a)

```
int a32[N];
if((N % 6) == 0) {
    for(i=0; i < N; i += 6) {
        w = a32[i];
        ... = (w & 0x0000001F):5;
        ... = ((w & 0x000003E0) >> 5):5;
        ... = ((w & 0x00007C00) >> 10):5;
        ... = ((w & 0x000F1000) >> 15):5;
        ... = ((w & 0x01F00000) >> 20):5;
        ... = ((w & 0x3E000000) >> 25):5;
    }
}
```

(b)

```
...
for(int i=3; i<N; i++) {
    y[i] = y[i-1]*w1+
           y[i-2]*w2+
           y[i-3]*w3;
}
...
```

(a)

```
int D3, D2=0, D1=0;
...
for(int i=1; i<N; i++) {
    D3 = D2;
    D2 = D1;
    D1 = y[i-1];
    If(i>2)
        y[i] = D1*w1+D2*w2+D3*w3;
}
...
```

(b)

6.3.1. The CHIMAERA-C Compiler. The CHIMAERA-C compiler [Ye et al. 2000a] extracts sequences of instructions from the C code, each one forming a region of up to nine input operands and one output. The goal of this approach is to transform small acyclic sequences of instructions, without side-effect operations, in the source code into special instructions implemented in the RPU (which in this case is a simple reconfigurable functional unit) [Hauck et al. 2004]. This approach aims to achieve speed-ups of the overall execution. As the compiler only considers *reconfigware* compilation for the previously described segments of code, it is inappropriate for used as a stand-alone *reconfigware* compiler.

6.3.2. The Compiler for Garp. An ANSI C compiler for the Garp architecture, called *garpcc* [Callahan et al. 2000], was developed at the University of California at Berkeley. It uses the hyperblock[13] [Mahlke et al. 1992] to extract regions in the code (loop kernels) suitable for *reconfigware* implementation, and transforms "if-then-else" structures into predicated forms. The basic blocks included in a hyperblock are merged and the associated instructions are represented in a single DFG, which explicitly exposes the operation-level fine-grained parallelism [Callahan and Wawrzynek 1998]. The compiler integrates *software pipelining* techniques [Callahan and Wawrzynek 2000], adapted from previous work on compilation, to VLIW processors. The compiler uses predefined module generators to produce the proprietary structural description. The *garpcc* compiler relies on the *Gama* tool [Callahan et al. 1998] for the final mapping phases. *Gama* uses mapping methods based on dynamic programming and generates from the DFG a specific array configuration for the reconfigurable matrix by integrating a specific placement and routing phase.

The ideas of the *garpcc* compiler were used by Synopsys in the Nimble compiler [Li et al. 2000]. As *garpcc*, Nimble also uses as a front-end the SUIF framework [Wilson et al. 1994]. The target of the Nimble compiler are devices with an embedded CPU tightly coupled to an RPU. The compiler uses profiling information to identify inner loops (kernels) for implementation on a specific datapath in the RPU, aiming at the overall acceleration of the input program. *Reconfigware*/software partitioning is automatic. The *reconfigware* segment is mapped to the RPU by the ADAPT datapath compiler. This compiler uses module descriptions for generating the datapath, and generates a sequencer for scheduling possible conflicts, for example, memory accesses, and orchestration of loops. The compiler also considers floor-planning and placement during this phase. The datapath structure with placement information is then fed to the placement and routing FPGA vendor tool to generate the bitstreams.

6.3.3. The NAPA-C Compiler. The NAPA-C compiler uses a subset of C with extensions to specify concurrency and bit-widths of integers [Gokhale and Stone 1998]. It targets

[13]A hyperblock represents a set of contiguous basic blocks. It has a single point of entry and may have multiple exits. The basic blocks are selected from paths commonly executed and suitable for implementation in *reconfigware*.

)

)

)

)