PRMDL: a Machine Description Language for Clustered VLIW Architectures

Andrei Terechko, Evert-Jan Pol, Jos van Eijndhoven
Philips Electronics Nederland B.V.
{andrei.terechko, evert-jan.pol, jos.van.eijndhoven}@philips.com

Abstract

A compiler-simulator framework must be retargetable to enable platform-based processor design as well as proper processor architecture design space exploration. This paper describes the design decisions taken for the retargetability mechanism of the Philips Research compiler-simulator framework driven by a central machine description file. The format of the machine description file plays an important role in defining the scope of retargetability of a compiler-simulator framework. The machine description format PRMDL used in Philips Research supports a wide variety of VLIW architectures. In particular, PRMDL is capable of expressing clustered architecture features such as incomplete bypass networks, multiple register files, along with functional units shared or distributed among multiple issue slots, diverse conditional operation mappings, and more. The structure of PRMDL features separate software and hardware views on a processor. This insures robustness of retargetability built into tools across several processor generations.

1. Introduction

As time-to-market requirements demand faster design cycles, more programmable components are introduced in embedded systems. We are exposed to a substantial shift of system functionality from hardware to software. In order to use massive hardware parallelism efficiently, an increasingly complicated compilation trajectory should be developed. However, its complexity should not affect system development time. Hence, traditional compilers, which must be rewritten for every new processor generation (see Figure 1), do not suffice any more.

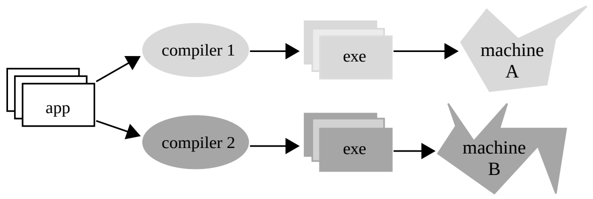

Figure 1. A traditional compilation trajectory

This implies that the compilation trajectory must be able to change its target machine quickly and easily, which can be achieved by parameterising a compiler toolchain (see Figure 2). The compiler tools read target machine parameters from a machine description file and adjust their processing accordingly. To retarget the whole compiler toolchain one only needs to change the target processor description in the machine description file. This scheme enables substantial reuse of the complicated compiler software [2].

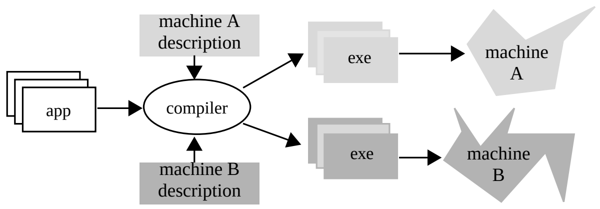

Figure 2. A retargetable compilation trajectory

In the design of advanced processors like the new 64-bit TriMedia CPU64 [3] many decisions must be taken. For example, one of the most challenging tasks in the design of the CPU64 was studying the impact of the quantity and slot assignment of the functional units. The size of the design space for this task had 10^{15} possible solutions [1]. Such tasks need systematic exploration of the design space with numerous iterations over processor instances in order to provide quantitative data that design decisions can be based on. For this purpose a retargetable design space exploration (DSE) framework including both retargetable simulator and compiler should be deployed (see Figure 3). Varying parameters in the machine description file quickly retargets the machine-independent compiler and simulator without even recompiling the framework.

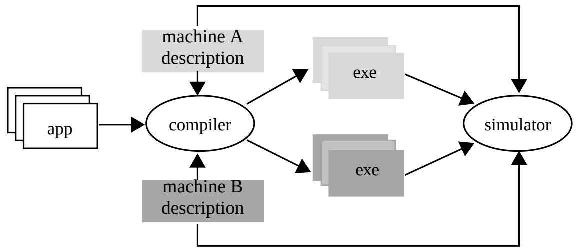

Figure 3. A retargetable DSE framework

The format of the machine description file reflects the scope of framework retargetability and thus decides between fixed elements and configurable elements of the architecture template. Consequently, the format has a great influence on reuse of the framework across multiple processor generations, the maintenance of the framework, as well as on framework performance characteristics.

2. Related work

Existing machine description languages, in which target processor parameters are expressed, differ considerably. In order to put PRMDL (Philips Research Machine Description Language) in perspective with other languages, one can consider the classification described in [4], which presents three categories. Behavioural machine description languages (nML [8], ISDL [7], Insulin, etc.) describe a processor in terms of its instruction set. Structural machine description formats (MIMOLA [9], MLRISC, etc.) primarily focus on a structural model of the architecture. Mixed-level languages (PRMDL, EXPRESSION [4], HMDes [5], LISA [6], etc.) combine both structural and behavioural views and drive both compiler and simulator (see Figure 3).

Compared to HMDes, the PRMDL format is simpler and requires less programming effort than HMDes. HMDes captures constraints between operations with explicit reservation tables, using a hierarchical description for compactness. While PRMDL aims primarily all at compilers and simulators for TriMedia CPUs, HMDes appears to suit better research-oriented architecture explorations.

EXPRESSION has syntax simplicity and coverage of architectures similar to PRMDL. Among its strong points are the explicit specification of the memory subsystem and the graphical user interface. In EXPRESSION, like in PRMDL, the reservation tables for the processor operations are derived from the processor structural description. EXPRESSION features plain LISP-like syntax and relative ease of modifications. However, having about one thousand operation mappings in the TriMedia compilation trajectory, more concise and legible PRMDL description of mappings is advantageous. On top of that, the PRMDL syntax allows mapping across architectures with different data path widths and various conditional mappings (see Section 4.4).

3. Physical and virtual machines

PRMDL features explicitly separate software and hardware views on the processor (see Figure 4). The physical machine constituting the hardware view accommodates all parameters of the processor hardware architecture, such as register file and issue slot parameters. The virtual machine constituting the software view contains the programming model of the processor. Using software operations from the virtual machine, the application programmer writes code in C. During the compilation of the application the software operations are mapped on hardware operations from the physical machine. The virtual machine operation to physical machine operation mappings are denoted by arrows in Figure 4.

The virtual machine remains constant throughout several generations of the processor hardware, which preserves C source-level compatibility. One virtual machine in Figure 4 provides the programmer with a uniform software interface to multiple physical machines. Changes in the processor hardware influence not the C code but the machine description file. For example, if a hardware operation is left out in the next processor generation, only rewriting the mappings in the machine description file for software operations mapped onto the missing hardware operation is required.

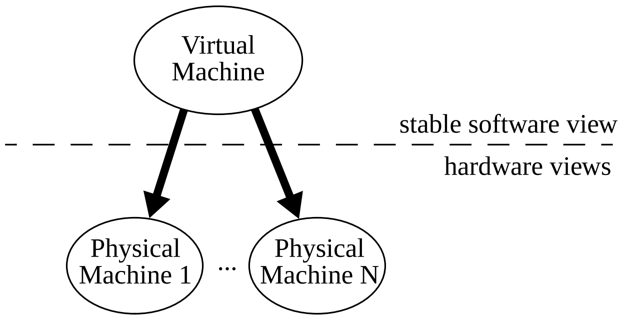

Figure 4. Separate software and hardware views

The software operation descriptions in a virtual machine carry operand and result type information, which enables type-checking in C sources, while the hardware operations in a physical machine are type-independent, so that the same hardware operation can be used with different argument and result types. The instruction set architecture of the virtual machine shows an orthogonal set of operations over data types, which simplifies programming. On the other hand, the physical machine instruction set is reduced and reflects the processor hardware operations, which are sometimes type independent. The explicit separation of the physical machine and virtual machine also aids the maintenance of the large sets of hardware and software operations in a compiler-simulator framework.

4. PRMDL overview

The structure of the PRMDL format is as follows:

```
MACHINE_DESCRIPTION
DECLARATION
        TYPES
        RANGES
        SIDE_EFFECTS
        DATA_PATH_POINTS
PHYSICAL_MACHINE (* Hardware view on the CPU *)
        STATE
        SLOTS
        FUNCTIONAL_UNITS
        DATA_PATHS
        PHYSICAL_OPERATIONS
VIRTUAL_MACHINE (* Software view on CPU *)
        VIRTUAL_OPERATIONS
        CODE_CONVENTIONS
MAPPING_SECTIONS
```

The next four sections will elaborate on this structure.

4.1 Declaration

Essentially, the DECLARATION section is intended to improve legibility (types and ranges) and flexibility (side-effects) of the machine descriptions, as well as to ensure robust consistency checks. The TYPES section enumerates possible operand and result types of a virtual operation. The RANGES section describes integer ranges used in the descriptions of operation immediates, conditional mappings, and code convention clauses. The SIDE_EFFECTS list declares the side-effect hierarchy used by a compiler for generating ordering constraints between virtual operations. The DATA_PATH_POINTS section contains data path points that are not declared in the port sections of the slots and register files, but still designate resource conflicts in the described architecture. A declaration example:

```
DECLARATION
TYPES
        vec64sb,                    (* 64-bit signed byte vector *)
        vec64ub;                    (* 64-bit unsigned byte vector *)
RANGES
        pi8by2:     -128 TO 126 STEP 2,     (* a range for an immediate *)
        pu3p2:      POWER2 (0 TO 3);        (* a non-linear range *)
SIDE_EFFECTS
        pcsw, pcsw.intround, pcsw.fpflags;  (* a side-effect hierarchy *)
DATA_PATH_POINTS
        common_bus, switch;                 (* extra data path points *)
```

The range *POWER2(0 TO 3)* denotes the integer set {0,1,2,4,8}. The side-effect hierarchy (see in detail Section 4.2) in this example specifies two side-effects *pcsw.intround* and *pcsw.fpflags,* that can also be addressed together as *pcsw. pcsw* denotes the Program Control and Status Word register in TriMedia processors, which contains processor control and status bits such as floating point exception flags (*pcsw.fpflags*), integer round mode bits (*pcsw.intround*), etc.

4.2 Physical Machine

The Physical Machine section contains the processor hardware model, including a hardware operations list. The STATE section describes processor resources holding its state (primarily register files, but also other types of processor memory). In the STATE section one can describe register file properties, such as register width, number of registers, constant registers, access time, read/write ports, overlapping, access type (e.g. random, FIFOs, LIFOs), look-up tables, etc.

```
STATE
rf0     WIDTH 64                        (* register file rf0 *)
        NUMBER 128                      (* 128 64-bit registers *)
        INDEX_RANGE 0 TO 127            (* index range is from 0 to 127 *)
        ACCESS_MANNER random            (* random access RF *)
        PORTS (wp0, wp1 -> gp0,rp0,rp1,rp2,rp3) ; (* write/read ports *)
```

The distribution of functional units among VLIW issue slots is described in the FUNCTIONAL_UNITS section. There are three types of functional units in PRMDL: an ordinary functional unit, which occupies one issue slot, a super functional unit, which occupies more than one issue slots [3], and a shared functional unit, which can be controlled via several issue slots.

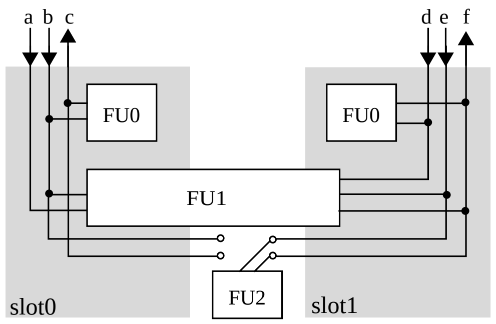

Figure 5. Functional unit types

The configuration in Figure 5 includes slot0 and slot1 with four functional unit instances: two FU0 of the ordinary type, one super functional unit FU1, and one shared functional unit FU2. A description of this structure in the PRMDL language is as follows:

```
SLOTS
        slot0 (a,b->c), slot1 (d,e->f);         (* two slots are defined *)
FUNCTIONAL_UNITS
FU0             (* two instances of an ordinary functional unit *)
        SLOTS  slot0(b->c), slot1(d->f);
        TIME_SHAPE (0->1)
        UNIT_OPERATIONS op1(0), op2(1);
FU1             (* a super functional unit *)
        SLOTS  slot0(a,b->) & slot1(d,e->f);
        TIME_SHAPE (0,0,1,1->2)
        UNIT_OPERATIONS op3(2),op4(3);
FU2             (* a shared functional unit *)
        SLOTS slot0(b->c) | slot1(e->f);
        TIME_SHAPE (0->9)
        UNIT_OPERATIONS op5(4);
```

The TIME_SHAPE clauses specify timing properties of functional unit operations. For example, the expression *TIME_SHAPE (0,0,1,1->2)* specifies that the first two arguments of an operation from the functional unit FU1 arrive in cycle 0 to the slot ports, the two others can be read in cycle 1, and the result is produced in cycle 2.

The DATA_PATH section specifies intra-processor connectivity. It primarily serves to define resource conflicts on the register file or slot ports or some abstract data path points defined in the DECLARATION section. The bypass networks, writeback-bus schedulers, and inter-cluster communication paths can all be described in the DATA_PATH section.

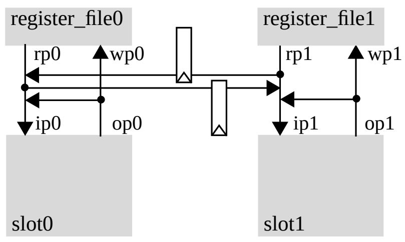

Figure 6. Various data paths

rp0, *rp1*, *wp0*, *wp1*, *ip0*, *ip1*, *op0*, *op1* in Figure 6 designate read, write, input, and output ports, respectively. A description of the processor data paths in Figure 6 is as follows:

```
DATA_PATHS
        rp0 -> ip0 DELAY 0;      (* a read bus *)
        rp1 -> ip1 DELAY 0;      (* a read bus *)
        op0 -> wp0, ip0 DELAY 0; (* a writeback bus and bypass *)
        op1 -> wp1, ip1 DELAY 0; (* a writeback bus and bypass *)
        rp0 -> ip1 DELAY 1;      (* inter-cluster communication *)
        rp1 -> ip0 DELAY 1;      (* inter-cluster communication *)
```

The PHYSICAL_OPERATIONS section contains operation names, guards, arguments, results, properties, and side-effects. All this information is combined in operation signatures. The signature inputs can be issue slot operands designated by '*', immediate arguments designated by the range name that the immediate must fit, and read side-effects. The outputs can be either issue slot operands or write side-effects. An optional guard '*?' specifies that the operation is conditional. For example, the line

```
PHYSICAL_OPERATIONS
        SIGNATURE        (*? *,*->*,*,pcsw.fpflags) pop1, pop2;
```

defines properties of the physical operations *pop1* and *pop2*, which are guardable, take two operands, return two results, and write to *pcsw.fpflags* (a side-effect).

In PRMDL terms, side-effects are changes in the machine state apart from direct input/output data flow (e.g. in a register file) caused by operations. Examples of side-effects are machine flags affected by floating point operations, a program counter affected by branch operations, memory changes affected by load/store operations, etc. The compiler can use them to generate sequential ordering constraints for operations. Initially, the side-effects are specified in the physical signatures, from which all physical operations, subsequently, inherit them. Virtual operations are translated into physical ones during the compilation process. Therefore they must inherit side-effects from the corresponding physical operations in order to enable the ordering constraints generation by the compiler front-end. The propagation of side-effects from physical to virtual operations ensures conciseness and consistency of the operation property descriptions.

4.3 Virtual Machine

The Virtual Machine section contains the programming model of the processor. The VIRTUAL_OPERATIONS section includes software operation signatures, which contain operation names, argument and result types, and operation properties. For example, the lines

```
VIRTUAL_OPERATIONS
        SIGNATURE (vec64sb,vec64sb->int64) COMMUTATIVE vop1,vop2;
```

describe commutative operations *vop1* and *vop2*, which take two operands of the type *vec64sb* and return a result of the type *int64*. The *vec64sb* and *int64* types must be declared in the DECLARATIONS section. The compiler front-end can use this type information in type checking and casting.

The CODE_CONVENTIONS section includes a list of compiler-oriented code conventions such as the return value register, the stack pointer register, global and local register pools, etc.

4.4 Mapping

The MAPPINGS_SECTION sections define operation transformations, capable of driving parameterised code selection at all compilation stages. The mappings can include conditional clauses, where an operation is mapped onto different groups of operations depending on a condition that should be matched by an immediate argument of the operation. The Mapping section syntax also allows defining instruction set transformations across architectures with different data path widths.

Each mapping section can include two types of mappings: conditional mappings and ordinary mappings. An ordinary mapping defines a transformation of a source operation into a set of target operations.

```
MAPPINGS
        vimm8 (i8 -> z) = pimm16 ((i8<<8)+i8 -> z); (* parameter expression *)
        sb_subsame (x,y -> z) = packsame_b (y ->A ), sub_b (x,A ->z );
```

The mappings can have temporary variables, parameter expressions, and references to processor registers. The PRMDL format is also capable of specifying mappings across architectures with different data path sizes (e.g. from a 128-bit CPU onto a 64-bit one). In order to do so PRMDL allows to address fractions of the arguments and results:

```
vadd128 (x,y->z) = vadd64 (x.0, y.0 -> z.0), vadd64 (x.1, y.1 -> z.1);
```

In this example *x.0*, *y.0*, and *z.0* refer to the lower 64 bits of the arguments and results, while *x.1*, *y.1*, and *z.1* denote the upper 64 bits.

Conditional mappings describe a transformation of a source operation onto different sets of target operations depending on a condition, which is evaluated with a help of immediate arguments of the source operation. The possible condition types are the following:

1. fitting a declared range:

```
SWITCH vmul (x,y->z) =
        CASE y IN_RANGE POWER2(imm_range1)
                vshift(x,LOG2(y)->z);
```

This type of mapping is especially useful for custom operations with immediates, which can differ from processor to processor significantly. The code selection for these operations can be parameterised using such mappings.

2. fitting the range of an argument of a physical operation:

```
SWITCH vmul (x,y->z) =
        CASE y IN_RANGE OPERATION_RANGE (pimul)
                pimul;
        CASE x IN_RANGE OPERATION_RANGE (pimul)
                pimul (y,x->z);
        DEFAULT pmul;
```

The condition of this mapping is defined by an operation with an immediate argument rather than by the range of the immediate itself. In the example above, *vmul* is mapped onto *pimul* if *x* or *y* fits the range of an immediate argument of the operation *pimul*, otherwise it is mapped on *pmul*.

3. matching a pattern:

```
SWITCH vmul (x,y->z)
        CASE y FITS_EXPRESSION POWER2(p)+POWER2(q)
                pshift (x,p->a), pshift (x,q->b), padd (a,b->z);
        DEFAULT pmul;
```

This mapping can be used to define code selections based on a pattern matching condition. The example, for instance, defines the mapping of the *vmul* operation onto the *pshift*, *shift*, and *padd* operations if there exist integer *p* and *q* such that $y = 2^p + 2^q$. This mapping type, however, doesn't support simultaneous equations and available operations in the pattern are limited to +, -, *, /, *POWER2*, *LOG2*, and *NOT*.

5. Conclusions

This paper describes a powerful compiler-simulator retargetability mechanism, which enables template-based processor design and allows for fast and vast design space explorations for future clustered VLIW processors. The mechanism is controlled by framework parameters stored in a central machine description file. The key features of the machine description file format PRMDL are as follows:

• *Explicit separation of compiler front-end (Virtual Machine) and back-end (Physical Machine) instruction sets, which provides better source-code compatibility*

• *Support for forthcoming clustered architectures with multiple register files and incomplete connectivity*

• *C types of arguments in virtual operation signatures help the compiler to do type checking and casting*

• *Side-effects in virtual operation signatures help the compiler to generate optimal ordering constraints for operations*

• *Supported diversity of types of local storage (random access register files, LIFOs, FIFOs, etc.)*

• *Conditional mappings allow full parameterisation of code selection in the compiler front-end*

• *Mapping across architectures with different data path sizes ensures strong processor family compatibility*

• *Parameter expressions in mappings allow arithmetic operations on immediates in parameterised code selection*

Acknowledgments

A number of dedicated architects and engineers at Philips Research Eindhoven have contributed to the design of the PRMDL machine description format and its integration in the compiler-simulator framework. These people include J.v. Eijndhoven, E.J. Pol, R. Baas, J. Leijten.

6. References

[1] G.J. Hekstra, G.D. La Hei, et al. “TriMedia CPU64 Design Space Exploration”. In Proceedings International Conference on Computer Design, Austin, Texas, October 1999, pp. 599-606.

[2] E.J.D. Pol, B.J.M. Aarts, et al. “TriMedia CPU64 Application Development Environment”. In Proceedings the International Conference on Computer Design, Austin, Texas, October 1999, pp. 593-598.

[3] J.T.J. van Eijndhoven, F.W. Sijstermans, et al. “TriMedia CPU64 Architecture”. In Proceedings of the International Conference on Computer Design, Austin, Texas, October 1999, pp. 586-592.

[4] P. Grun, A. Nicolau, et al. “Expression: a language for architecture exploration through compiler/simulator retargetability”. In Proceedings of the Design Automation and Test in Europe, Paris, France, March 1999.

[5] J. C. Gyllenhaal et al. “The MDes user manual“. Technical report, http://www.trimaran.org, 1998.

[6] V. Zivojnovic, S. Pees, and H. Meyr. “LISA - machine description language and generic machine model for HW/SW co-design” In Proceedings of the IEEE Workshop on VLSI Signal Processing, San Francisco, October 1996.

[7] G. Hadjiyiannis, S. Hanono, and S. Devadas. “ISDL: An instruction set description language for retargetability“. Proceedings of Design Automation Conference, Anaheim, CA, May 1997.

[8] M. Freericks. “The nML machine description formalism“. Technical Report TR SM-IMP/DIST/08, TU Berlin, Computer Science Dept., July 1993.

[9] R. Leupers and P. Marwedel. “Retargetable code generation based on structural processor descriptions“. Design Automation for Embedded Systems, vol. 3, no. 1, 1998.