Reprinted from *Workshop on Compiler Support for System Software*, February 1996.

Reusable Application-Dependent Machine Descriptions

Mark W. Bailey
Jack W. Davidson[†]

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

Abstract

The proliferation of high-performance microprocessors in recent years has made the development of systems software, such as compilers, assemblers, linkers, debuggers, simulators, and other related tools, more challenging than ever. Despite their wide use in retargetable compilation systems, machine descriptions have seldom been used in other systems software. One reason is that machine descriptions tend to be application dependent. In this paper, we show how software can share descriptions that still contain application dependencies by using a new description framework called CSDL.

1 Introduction

For many years, compilers have used machine descriptions to capture details about the target machine. By using machine descriptions, target-specific information is isolated from the rest of the implementation so that it may easily be examined and changed. Despite their broad use in compilers, machine descriptions have seldom been used by other systems software such as linkers, debuggers, profilers, and simulators. For the most part, where machine descriptions have been used, new systems were developed rather than borrowing the technology—as Larus and Schnarr did [LS95]—from an extant description system. A primary motive for this action is that machine descriptions are application dependent. That is, inherent in the way a description is written is the purpose for which the application will use the information. We believe that most useful machine descriptions must contain some application-dependent aspects. In this paper, we present a system for building computing system descriptions that are application dependent, but can still be shared among many applications.

1.1 Application Independence

The use of a machine description can significantly reduce the retarget-time of an application [Wic75, Cat78, GH84, Dav85, BHE91, RF95]. However, with each retarget of the application, a description for the new target machine must be written. For an application of any substance, this itself can be a daunting task. There are three sources of difficulty:

1. Information about the machine must be found, encoded using whatever description technique is used, and must be tested, verified, and debugged to ensure accuracy. For some

[†]Authors' e-mail addresses: `mark@virginia.edu`, `jwd@virginia.edu`.

machines, finding the information is itself difficult. For some applications, the sheer volume of information to be encoded is a significant obstacle.

2. A description system that is tailored for a particular application usually contains bias toward that application. Thus, for example, a retargetable compilation system may include a machine description facility. This facility may require that information be encoded in a particular way, or that only some information be encoded. Typically, only an expert familiar with the compiler can write such a description though the concepts that are described do not require compiler expertise to understand.
3. Because the application does not share a common description format with other applications, one can be certain that there is not already a description available for one's use.

Using a common description format that contains no application bias eliminates these three sources of difficulties. Such a description facility is called *application independent*. Obviously for an application-independent description it may at least be possible that the description already exists for the new target machine (source 3). Further, no knowledge of a particular application is required to successfully write a description (source 2). Thus any computer professional who is familiar with the machine should be qualified to write a description. Finally, if an application-independent description system becomes widely used, finding information about a target machine will become easier since computer manufacturers could supply documentation about the machine in the form of a system description (source 1).

2 A Framework for Building Machine Descriptions

Our system extends previous work in machine descriptions in three key ways: abstraction level, extensibility, and modularity. Because this new description system widens the abstraction level of machine descriptions, we call them *computing systems descriptions* to reflect their broader applicability [BD95c]. Our description system, called CSDL (*Computing System Description Language*), is a framework for developing more thorough, complete descriptions of target machines for use in retargetable systems software implementations.

As shown in Figure 1, CSDL is a framework that divides computing-system information into modules, or components. One component is distinguished from all the others: it contains the core description for the system. The *core* contains the description of the instruction set of the machine. As its name implies, it is required to be present in all CSDL descriptions, while the other components may be optionally added or removed. The description of the instruction set, which is needed in nearly all systems software, gives an otherwise amorphous system a coherent structure. Unlike the optional components, where nothing but the most minimal structure is imposed, the core's structure, or format is defined by CSDL. By defining its structure, we can insure that the most-widely-used component of the system is application independent, thereby promoting its adoption by many applications.

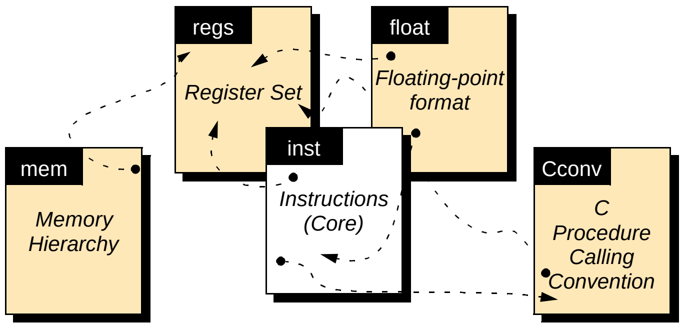


Figure 1: Computing system description framework

In addition to the core, CSDL incorporates application-defined components. A component provides additional information that is of interest to some, but not necessarily all, systems software. Since a component is application defined, it can present the information at the level of abstraction that is most appropriate for the defining application. Examples of components include pipeline and memory descriptions for different implementations of the same architecture, object file formats used by the assembler and linker, and descriptions of high-level-language procedure calling conventions.

By providing modular descriptions, applications only need to examine the parts they are concerned with. Thus, descriptions need not be “complete” to be valid or useful. Different machine models might share certain parts of a description, but distinct models might have different pipeline descriptions or memory interface descriptions. Modularity also supports ease of modification. A new model of a machine might have a different pipeline, but the ISA (*Instruction-Set Architecture*) and calling conventions likely remain the same. Thus, only the part of the description that involves the pipeline needs to be modified. Similarly, modularity helps keep the various pieces of a system description concise. The component that describes the pipeline does just that, and nothing else.

Because CSDL descriptions are modular, significant flexibility is available to each application. The disadvantage of dividing descriptions into smaller, more manageable pieces is that this isolates each module. Without additional support, each component is likely to encounter the same pitfalls that many modular systems have: repetition among modules, and inconsistency between modules. To counter this tendency, CSDL has several mechanisms that aid in preventing inconsistency and repetition its modules. These mechanisms are the *glue* that holds CSDL descriptions together, and give them their descriptive power.

2.1 Linked Values

A disadvantage of dividing descriptions into modules is that it is common for two or more modules to need access to the same information. To promote the sharing of common information between modules, CSDL provides a mechanism for introducing *linked values*.

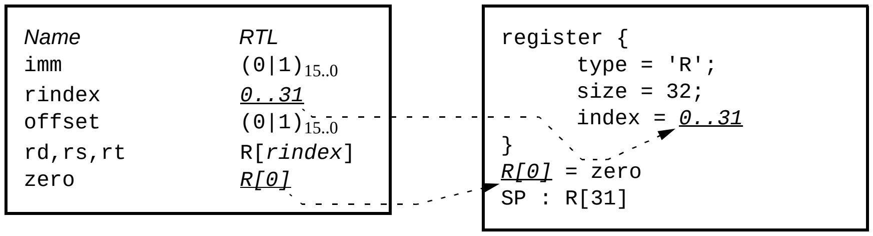
Any module may introduce a name/value pair. For example, a register description would want to be able to introduce names and values for the following registers: the program counter, the stack pointer, a register that is always zero, and the register that contains a routine’s return address. Using CSDL’s naming system, the register description can easily provide names and values for each of these registers. These names can then be subsequently referenced in other modules. Although the convention about which register contains the stack pointer must be written down, it is only written down once. The value can then be propagated though the system to the other modules using links.

Figure 2 demonstrates module linking. A register description excerpt, shown in Figure 2b, defines the valid register indices as well as defining register zero (`R[0]`) as always storing the value zero. An instruction description excerpt, shown in Figure 2a, contains references to these two values. To accurately define the valid instructions for the machine, the instruction description must know which register indices are valid. The instruction description refers to the valid register indices by name. Changes to the register description are immediately reflected in each referencing module.

The definition of values and their successive reference in other modules creates a web of information. These linked values are hypertext values that facilitate navigation throughout the description system. They also represent the relationship between objects in different modules. The reader of a description can better understand the interaction between objects in different description components because of the explicit representation of value references.

2.2 Application Annotations

A primary shortcoming of previous machine description techniques is that they present information in an application-dependent way. While the inclusion of application-specific information



```
Name            RTL
imm             (0|1)15..0
rindex          0..31
offset          (0|1)15..0
rd,rs,rt        R[rindex]
zero            R[0]
```

```
register {
      type = 'R';
      size = 32;
      index = 0..31
}
R[0] = zero
SP : R[31]
```

Figure 2a: Instruction module excerpt **Figure 2b:** Register module excerpt

Figure 2: Linked values

makes the descriptions easier for the particular application to use, it frequently makes the descriptions useless for other purposes. CSDL provides *application annotations* to reconcile these differences.

Annotations are pieces of information that are attached to existing descriptions for an application. Annotations are tagged as belonging to a particular application. When that application is viewing the description, the annotations appear as part of it, whereas when other applications view the description, the annotations are not present. Annotations can be thought of as an overlay, as shown in Figure 3, which an application places over a module. Application developers can scribble whatever information they wish without influencing other applications that are using the same module.

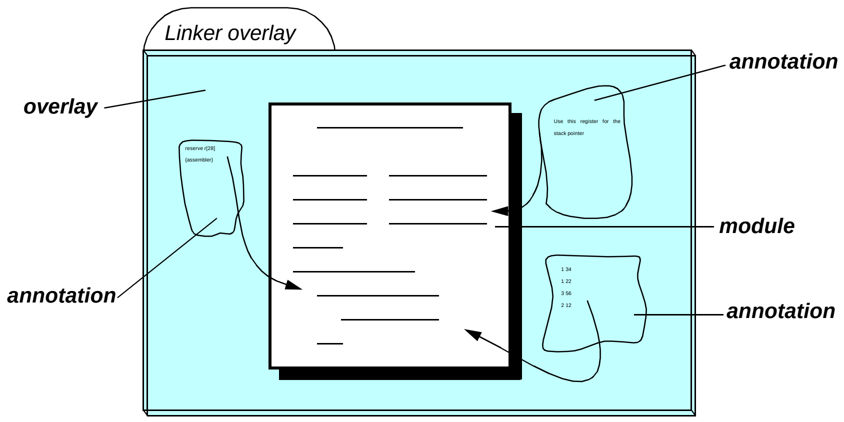


Figure 3: An application's annotation overlay

To illustrate the use of annotations, consider a compiler that uses information in the core instruction module for generating assembly language instructions for the MIPS R2000. The compiler needs to generate an instruction to move a value from one register to another. However, the MIPS does not explicitly provide a register-to-register move instruction. The RTL [DF84] instruction description is pure[1], that is, it contains no synthetic instructions. Thus, no move instruction is listed. On the MIPS, a logical-or instruction is used with register R[0] as the second operand to synthesize the move instruction. If the compiler cannot glean this information from the description, an annotation can be attached to the OR instruction, as shown in Figure 4, to indicate that a specific form may be used to achieve the move.

1. A pure description contains no synthetic or artificial instructions. We forbid the use of such impurities so that applications that depend on pure descriptions are not misled.

```
Name          RTL
arithmetic    rt ← rs op imm
              rd ← rs | rt ● - - - ->  rd ← rs ≡ rd ← rs ∨ zero
```

Figure 4: A CSDL annotation

2.3 Module Aspects

A concept closely related to annotations are module aspects. Although annotations may be used to attach small amounts of information to selective parts of a module, for situations where more significant additions to modules are necessitated, CSDL provides *module aspects*.

A compiler's instruction description may include an enormous amount of information: semantics of the instructions, assembler mnemonics, binary format, instruction costs, pipeline scheduling information, *etc*. However, much of this information is not contained in the core description for instructions. Many applications may only have use for the semantics of the instructions and the assembler format. Any part of the description can be tagged as an *aspect*. An aspect is another view of an object in the description. The aspect is used to selectively filter the descriptions. Just as annotations can be viewed as overlays, aspects can as well. However, unlike an annotation overlay that is tagged for a particular application, an aspect overlay may be made available for use by any application. Thus, if a compiler is only interested in the semantics, instruction cost, and binary format, only those overlays are taken from the overlay "library" and placed over the module. This provides a mechanism for components to have many facets that are used by many applications.

Figure 5 illustrates the use of aspects. Here, the core description is augmented with two aspects: an assembly language aspect and a binary format aspect. Although aspects are usually keyed using color, here they appear as shaded boxes. The assembly aspect is shown in the left box, while the binary format aspect appears on the right. In each case, each element of an aspect has a corresponding element in the original module. So, for our example, each element of the binary format and assembly language aspects is associated with an instruction, or other object in the instruction description.

Name	*RTL*	assembly	binary
rd,rs,rt	R[*rindex*]	$rindex	rindex
zero	*R[0]*	0	0
op	+	addi	ADDI
	$+_u$	addiu	ADDIU
arithmetic	*rt ← rs op imm*	op rt,rs,imm	[op,rs,rt,imm]
	rd ← rs op1 rt	op1 rd,rs,rt	[SP,rs,rt,rd,0,op1]

Figure 5: Assembly language and binary format aspects of instructions

3 Applications

CSDL has been designed as a multi-application description framework from the start. Applications can use or extend existing CSDL modules, or add additional modules to suit their needs. In this section, we present several examples of how different systems software applications could use CSDL to build descriptions that can be shared among many applications.

3.1 Binary Translation

Binary translators take executable programs for a source machine and translate them to executable programs for a target machine. This application requires information about the binary instruction format for two machines. At first, it may seem that an implementation could simply take two instruction descriptions and automatically derive the translation from one format to the other.

However, binary translation is not such a simple problem. Often, the necessary translation from one instruction-set to another is not readily apparent; human intervention is necessary to glean the proper translation.

Consider a translator that converts SPARC executables into MIPS executables. An alternative approach to the one described above would be to annotate the SPARC description with the necessary information to perform the translation to MIPS instructions. This can be accomplished using a CSDL SPARC description by adding a MIPS translation aspect to each of the SPARC instructions. Figure 6 shows an excerpt from such solution.

rt ← *rs* + *imm*$_{13}$	{ emit("addi *rt*,*rs*,*imm*$_{13}$"); }
rt ← R[*rs* + *rx*]	{ emit("add *tmp*,*rs*,*rx*";emit("lw *rt*,0(*tmp*)"); }
R[*rt* + *rx*] ← *rs*	{ emit("add *tmp*,*rt*,*rx*";emit("sw *tmp*,0(*rs*)"); }
ST ← *label*, *n*	{ Call_Fixup_and_Emit(*label*, *n*); }
rt ← HI[*const*$_{22}$]	{ high16 = *const*$_{22}$ >> 6;
	low6 = (*const*$_{22}$ & 0x3f << 13);
	emit("andi *rt*,*rt*,0x1fff");
	emit("ori *rt*,*rt*,low6");
	emit("lui *rt*,high16"); }

Figure 6: Specifying binary translation using a CSDL aspect

To each SPARC instruction, a piece of C code is attached to perform the necessary translation. For example, the SPARC contains a load instruction that uses an indexed addressing mode. Since the MIPS doesn't contain an indexed mode, one is synthesized using two instructions: an add and a load word instruction. The MIPS and SPARC also differ on the size of their immediate operands. On the SPARC, an immediate value is 13 bits, while on the MIPS, an immediate value is 16 bits. Thus, SPARC load-immediate instructions are trivial to translate to MIPS instructions since SPARC immediate values are always smaller. However, to form a 32-bit constant, the SPARC has a sethi instruction that loads the high 22 bits of a register. When such an instruction is encountered, several MIPS instructions must be emitted to synthesize the load.

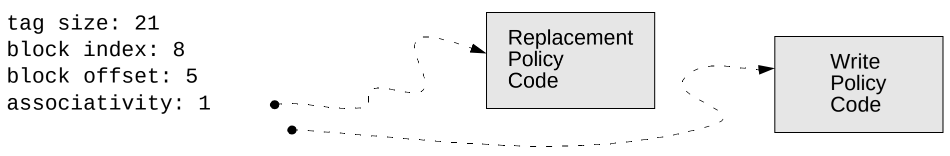
It is possible that one could automatically generate the translations described above. However, for complex instructions, such as CALL, it is unlikely that an automatic process will succeed. Such instructions assume other processor state, such as a particular stack layout or register usage. However, by attaching small portions of C code, we can easily reference external hand-written functions that perform these complex translations. Other situations, such as operating system traps, exception handlers, and instructions that use special-purpose registers can be handled in a similar way.

By providing extensions to modules, CSDL permits applications to embed application-dependent information into otherwise application-independent descriptions. In our example, the translations are application dependent, however, the instruction description is not. Although the description contains the binary translator's translation methods, CSDL's aspect mechanism makes it easy for other applications to filter out these instruction translation aspects and make use of the rest of the instruction description.

3.2 Cache Simulation

Another example of an application that requires target-machine information is a cache simulator. A cache simulator requires two kinds of information about the machine: instruction binary formats, and cache characteristics. The instruction formats are used to extract addresses from the instruction trace. We do not consider the encoding of binary formats here, since their inclusion in a CSDL instruction description is straightforward. A cache description, however, presents an interesting problem. A cache's structure can be easily characterized by giving values to the following quantities: tag size, block index size, block offset size and set associativity. For many applications, this information is sufficient. However, for a cache simulator to accurately predict the cache hit rate, and characteristics about cache misses, additional information is required.

To determine cache performance, the cache replacement policy and the cache write policy must be known. In both cases, many variants exist. Further, it is not immediately apparent how such policies could be described in an application-independent way. An alternative is to provide code that implements each of these policies for the cache simulator. This code, which is clearly application-dependent is also cache-dependent. Thus, it seems only natural to include it with the other characteristics of the cache's design. As shown in Figure 7, the cache line replacement and write policies are included as annotations in a cache description that might already be used by a compiler. This makes it possible for the compiler and the simulator to share common information, and allow the simulator to annotate the description without having adverse effects on the compiler's implementation.



```
tag size: 21
block index: 8
block offset: 5
associativity: 1
```

Figure 7: An example cache description for the DEC Alpha AXP 21064

3.3 Other Applications

We have already developed several other CSDL components. These include a component for describing floating-point representations [BD95a] and procedure calling conventions [BD95b]. In both components, we have succeeded in making the descriptions application independent. We briefly describe them here.

A component of CSDL not found in any other description system is our description of floating-point representations. Our floating-point descriptions provide the information necessary to determine a floating-point number's value from its encoding. The description, which is declarative in nature, does not presuppose its use for a particular purpose. Among other purposes, these descriptions may be used for:

- automatically generating conversion routines for cross compilers,
- floating-point constant folding for cross compilers,
- transmission of floating-point values between heterogeneous architectures, and
- automatic generation of I/O routines for high-level language run-time libraries.

In addition to the floating-point format description, we have developed a description language for procedure calling conventions. The procedure calling convention, which dictates how information is transmitted between procedures, must be known to any application that manipulates procedure calls. We have used these descriptions to automatically generate instruction sequences for procedure calls in a compiler [BD95b] and to automatically construct target-specific test suites for applications that manipulate procedure calls [BD96]. These descriptions can further be used in other applications such as debuggers and linkers.

4 Summary

In this paper, we have presented a new framework for developing descriptions of computing systems. The CSDL system facilitates the construction of descriptions that can be shared among many software applications. As a goal, we would like to build application-independent descriptions. In practice, this is not always feasible. CSDL recognizes this fact and provides the appropriate mechanisms to software developers.

Because it is difficult, if possible, to anticipate all of the information that all applications will need about a target machine, there always will be a need to add information to existing descriptions. By introducing annotations, modules, and aspects, our description system makes it possible to make these necessary extensions—without impacting existing applications. When details about a target machine are missing from a description, an application can extend the description system in whatever way is most appropriate for the application's purposes.

Finally, choosing the CSDL system for parameterizing an application does not preclude the use of an already proven system of description. Instead, with few, or no modifications, an extant description can be integrated with CSDL, enhancing its descriptive capability and making it available for other applications to use and extend.

Acknowledgments

We thank Sanjay Jinturkar for his comments on earlier versions of this paper. This work was sponsored in part by National Science Foundation grant CCR-9214904.

References

[BD95a] Mark W. Bailey and Jack W. Davidson. Describing the representation of floating-point values. Technical Report CS-95-43, Department of Computer Science, University of Virginia, Charlottesville, VA 22903, September 1995.

[BD95b] Mark W. Bailey and Jack W. Davidson. A formal model and specification language for procedure calling conventions. In *Proceedings of the 22nd SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 298—310, January 1995.

[BD95c] Mark W. Bailey and Jack W. Davidson. Computing system descriptions for systems software. Technical Report CS-95-10, Department of Computer Science, University of Virginia, Charlottesville, VA 22903, February 1995.

[BD96] Mark W. Bailey and Jack W. Davidson. Target-sensitive construction of diagnostic programs for procedure calling sequence generators. To appear in *Proceedings of the ACM SIGPLAN' 96 Conference on Programming Language Design and Implementation*, May 1996. Also available as technical report CS-95-44, Department of Computer Science, University of Virginia, Charlottesville, VA 22903.

[BHE91] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The marion system for retargetable instruction scheduling. In *Proceedings of the ACM SIGPLAN ' 91 Conference on Programming Language Design and Implementation*, pages 229—240, June 1991.

[Cat78] Roderic G. G. Cattell. Using machine descriptions for automatic derivation of code generators. In *Proceedings Third Jerusalem Conference on Information Technology*, pages 503—507, 1978.

[Dav85] Jack W. Davidson. Simple machine description grammars. Technical Report CS-85-22, Department of Computer Science, University of Virginia, Charlottesville, VA 22903, November 1985.

[DF84] Jack W. Davidson and Christopher W. Fraser. Code selection through object code optimization. *ACM Transactions on Programming Languages and Systems*, 6(4):505—526, October 1984.

[GH84] Susan L. Graham and Robert R. Henry. Machine descriptions for compiler code generation: Experience since JCIT-3. In *Proceedings Ninth Jerusalem Conference on Information Technology*, pages 236—250, 1984.

[LS95] James R. Larus and Eric Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN ' 95 Conference on Programming Language Design and Implementation*, pages 291—300, June 1995.

[RF95] Norman Ramsey and Mary F. Fernández. The new jersey machine-code toolkit. In *1995 Usenix Technical Conference*, pages 289—301, January 1995.

[Wic75] John D. Wick. *Automatic Generation of Assemblers*. PhD thesis, Department of Computer Science, Yale University, New Haven, CT, 1975.