LLVA: A Low-level Virtual Instruction Set Architecture

Vikram Adve Chris Lattner Michael Brukman Anand Shukla Brian Gaeke
Computer Science Department
University of Illinois at Urbana-Champaign
{vadve,lattner,brukman,ashukla,gaeke}@cs.uiuc.edu

Abstract

A virtual instruction set architecture (V-ISA) implemented via a processor-specific software translation layer can provide great flexibility to processor designers. Recent examples such as Crusoe and DAISY, however, have used existing hardware instruction sets as virtual ISAs, which complicates translation and optimization. In fact, there has been little research on specific designs for a virtual ISA for processors. This paper proposes a novel virtual ISA (LLVA) and a translation strategy for implementing it on arbitrary hardware. The instruction set is typed, uses an infinite virtual register set in Static Single Assignment form, and provides explicit control-flow and dataflow information, and yet uses low-level operations closely matched to traditional hardware. It includes novel mechanisms to allow more flexible optimization of native code, including a flexible exception model and minor constraints on self-modifying code. We propose a translation strategy that enables offline translation and transparent offline caching of native code and profile information, while remaining completely OS-independent. It also supports optimizations directly on the representation at install-time, runtime, and offline between executions. We show experimentally that despite its rich information content, virtual object code is comparable in size to native machine code, virtual instructions expand to only 2-4 ordinary hardware instructions on average, and simple translation costs under 1% of total execution time except for very short runs.

1. Introduction

In recent years, traditional superscalar processors and compilers have had to resort to increasingly complex designs for small improvements in performance. This has spurred a wide range of research efforts exploring novel microarchitecture and system design strategies in search of cost-effective long-term solutions [5, 21, 29, 31, 33]. While the outcome is far from clear, what seems clear is that an extensive rethinking of processor design has begun.

Along with design alternatives for the microarchitecture and system architecture, we believe it is also important to *rethink the instruction set architecture* — software's sole interface to the processor. Traditional processor instruction sets are a strait-jacket for both hardware and software. They provide little useful information about program behavior to the execution engine of a processor, they make it difficult for hardware designers to develop innovative software-controlled mechanisms or modify instruction sets, and they make it difficult for compiler developers to design optimizations that require hardware support beyond what can be expressed via machine instructions. The fundamental problem is that the same instruction set (the hardware ISA) is used for two very different purposes: as the persistent representation of software, *and* as the interface by which primitive hardware operations are specified and sequenced.

1.1. Virtual Instruction Set Computers

As a step towards loosening these restrictions, several research and commercial groups have advocated a class of architectures we term Virtual Instruction Set Computers (VISC) [9, 14, 23, 32]. Such an architecture defines a virtual instruction set (called the V-ISA in Smith et al.'s terminology [32]) that is used by *all user and operating system software*, as illustrated in Fig. 1. An implementation of the architecture includes both (a) a hardware processor with its own instruction set (the implementation ISA or I-ISA), and (b) an *implementation-specific* software translation layer that translates virtual object code to the I-ISA. Because the translation layer and the hardware processor are designed together, Smith et al. refer to this implementation strategy as a *codesigned virtual machine* [32]. Fisher has described a closely related vision for building families of processors customized for specific application areas that maintain compatibility and performance via software translation [15].

At the most basic level, a VISC architecture decouples the program representation (V-ISA) from the actual hardware interface (I-ISA), allowing the former to focus on cap-

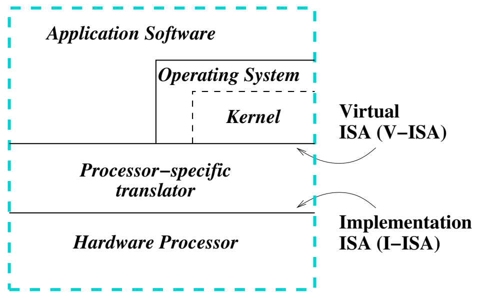

Figure 1. System organization for a virtual architecture. This is similar to Fig. 1(c) in [32] and Fig. 5 in [24].

turing program behavior while the latter focuses on software control of hardware mechanisms. This brings two *fundamental* benefits to the hardware processor design and its software translation layer:

1. The virtual instruction set can include rich program information not suitable for a direct hardware implementation, and can be independent of most implementation-specific design choices.
2. The I-ISA and its translator provide a *truly cooperative* hardware/software design: the translator can provide information to hardware through implementation-specific mechanisms and instruction encodings, while the hardware can expose novel microarchitectural mechanisms to allow cooperative hardware/software control and also to assist the translator.

These two fundamental benefits could be exploited in potentially unlimited ways by processor designers. Prior work [14, 32, 11] has discussed many potential hardware design options enabled by the VISC approach, which are impractical with conventional architectures. Furthermore, I-ISA instruction encodings and software-controlled mechanisms can both be changed relatively easily with each processor design, something that is quite difficult to do for current processors. Cooperative hardware-software techniques (including many examples proposed in the literature) can become much easier to adopt. Finally, external compilers can focus on machine-independent optimizations while the translator serves as a common back-end customized for the processor implementation.

The cost of this increased flexibility is the possible overhead of software translation (*if* it must be done "online"). Nevertheless, recent advances in dynamic compilation, program optimization, and hardware speed can mitigate the performance penalty, and could make this idea more viable today than it has been in the past. Furthermore, hardware mechanisms can be used to assist these tasks in many ways [14, 11, 19, 35, 27, 30].

1.2. Our Contribution: Design for A Virtual Instruction Set

Although virtual architectures have been discussed for a long time and real implementations exist (viz., IBM S/38 and AS/400, DAISY, and Transmeta's Crusoe), there has been little research exploring *design options* for the V-ISA. Both DAISY and Crusoe used traditional hardware ISAs as their V-ISA. The IBM machines do use a specially developed V-ISA, but, as we explain in Section 6, it is also extremely complex, OS-dependent, requires complex OS services for translation. Furthermore, as we explain in Section 4.1, the OS-dependent design enables a translation strategy integrated with the OS, which is impractical with OS-independent designs.

We believe a careful design for the V-ISA *driven by the needs of compiler technology*, yet "universal" enough to support *arbitrary* user and OS software, is crucial to achieve the full benefits of the virtual architecture strategy. A common question is whether Java bytecode (as suggested by Smith et al. [32]) or Microsoft's Common Language Infrastructure (CLI) could be used as a V-ISA for processors. Since a processor V-ISA must support *all external user software and arbitrary operating systems*, we believe the answer is "no". These representations are designed for a certain class of languages, and are not sufficiently language-independent for a processor interface. They include complex runtime software requirements, e.g., garbage collection and extensive runtime libraries, which are difficult to implement without operating system support. Finally, they are generally not well-suited for low-level code such as operating system trap handlers, debuggers, and performance monitoring tools.

This paper proposes a virtual ISA called Low-level Virtual Architecture (LLVA), and an accompanying translation strategy that does not assume particular hardware support. More specifically, this work makes three contributions:

- It proposes a V-ISA design that is rich enough to support sophisticated compiler analyses and transformations, yet low-level enough to be closely matched to native hardware instruction sets and to support all external code, including OS and kernel code.
- It carefully defines the behavior of exceptions and self-modifying code to minimize the difficulties faced by previous translators for DAISY [14] and Crusoe [11].
- It describes a translation strategy that allows offline translation and offline caching of native code and profile information (unlike DAISY and Crusoe), by using an OS-independent interface to access external system resources.

In other work, we are developing a complete compiler framework for link-time, install-time, runtime, and offline ("idle-time") optimization on ordinary processors, based on essentially the same code representation [26]. We discuss how the translation strategy proposed here can directly leverage those optimization techniques.

The virtual instruction set we propose uses simple RISC-like operations, but is fully typed using a simple language-independent type system, and includes explicit control flow graphs and dataflow information in the form of a Static Single Assignment (SSA) representation. Equally important is what the V-ISA does *not* include: a fixed register set, stack frame layout, low-level addressing modes, limits on immediate constants, delay slots, speculation, predication, or explicit interlocks. All of these are better suited to the I-ISA than the V-ISA. Nevertheless, the V-ISA is low-level enough to permit extensive machine-independent optimization in source-level and link-time compilers (unlike Java bytecode, for example), reducing the amount of optimization required during translation from V-ISA to I-ISA.

The benefits of a V-ISA design can only be determined after developing new processor design options and the software/hardware techniques that exploit its potential. Our goal in this work is to evaluate the design in terms of its suitability as a V-ISA. We have implemented the key components of the compilation strategy for SPARC V9 and Intel IA-32 hardware processors, including an aggressive link-time interprocedural optimization framework (which operates on the V-ISA directly), native code generators that can be run in either offline or JIT mode, and a software trace cache (for the SPARC V9) to support trace-based runtime optimizations. With these components, we address two questions:

- Qualitatively, is the instruction set rich enough to enable both machine-independent and dependent optimizations during code generation?
- Experimentally, is the instruction set low-level enough to map closely to a native hardware instruction set, and to enable fast translation to native code?

2. Design Goals for A Virtual ISA

Figure 1 shows an overview of a system based on a VISC processor. A VISC architecture defines an external instruction set (V-ISA) and a binary interface specification (V-ABI). An implementation of the architecture includes a hardware processor plus a translator, collectively referred to as the "*processor*." We use "*external software*" to mean all software except the translator. The translator is essentially a compiler which translates "virtual object code" in the V-ISA to native object code in the I-ISA.

In our proposed system architecture, *all external software* may only use the V-ISA. This rigid constraint on external software is important for two reasons:

1. To ensure that the hardware processor can evolve (i.e., both the I-ISA and its implementation details visible to the translator can be changed), without requiring any external software to be recompiled.
2. To ensure that arbitrary operating systems that conform to the V-ABI can run on the processor.

Because the primary consumer of a V-ISA is the software translation layer, the design of a V-ISA must be driven by an understanding of compiler technology. Most non-trivial optimization and code generation tasks rely on information about global control-flow, dataflow, and data dependence properties of a program. Such properties can be extremely difficult to extract from native machine code.

The challenge is to design a V-ISA that provides such high-level information about program behavior, yet is appropriate as an architecture interface for *all external software*, including applications, libraries, and operating systems. We propose a set of design goals for such a V-ISA:

1. *Simple, low-level operations that can be implemented without a runtime system*: To serve as a processor-level instruction set for arbitrary software and enable implementation without operating system support, the V-ISA must use simple, low-level operations that can each be mapped directly to a small number of hardware operations.
2. *No execution-oriented features that obscure program behavior:* The V-ISA should exclude ISA features that make program analysis difficult and which can instead be managed by the translator, such as limited numbers and types of registers, a specific stack frame layout, low-level calling conventions, limited immediate fields, or low-level addressing modes.
3. *Portability across some family of processor designs:* It is impractical to design a "universal" V-ISA for all conceivable hardware processor designs. Instead, a good V-ISA design must enable some broad class of processor implementations and maintain compatibility at the level of virtual object code for all processors in that class (key challenges include endianness and pointer size).
4. *High-level information to support sophisticated program analysis and transformations*: Such high-level information is important not only for optimizations but also for good machine code generation, e.g., effective instruction scheduling and register allocation. Furthermore, improved program analysis can enable more powerful cooperative software/hardware mechanisms as described above.
5. *Language independence*: Despite including high-level information (especially type information), it is essential that the V-ISA should be completely language-

independent, i.e., the types should be low-level and general enough to *implement* high-level language operations correctly and reasonably naturally.

6. *Operating system support*: The V-ISA must fully support arbitrary operating systems that implement the virtual Application Binary Interface (V-ABI) associated with the V-ISA (discussed in Section 3.1). Therefore, it must provide all the necessary low-level mechanisms such as traps, memory management, and low-level device I/O.

Note that high-level virtual machines such as JVM and CLI fail to meet goals #1 (they use complex, high-level operations with large runtime libraries), #5 (e.g., JVM and CLI are tailored for object-oriented languages with a particular inheritence model), and #6 (their complex runtime systems require significant OS support in practice). In contrast, traditional machine ISAs fail to meet goals #2 and #4, and satisfy #3 only to a limited extent (e.g., existing programs cannot exploit new hardware instructions or larger architected register files).

3. LLVA: A V-ISA for high performance

3.1. Components of the V-ISA

The LLVA Virtual Instruction Set is a source-language-neutral, low-level, orthogonal, three-address instruction set. Figure 2 shows an example C function and the corresponding LLVA code. The basic components of the instruction set are as follows.

Register Set and Memory Model LLVA uses an infinite, typed, register file where all registers are in Static Single Assignment (SSA) form [10] (described below). Registers can only hold scalar values, viz., boolean, integer, floating point, and pointer. This type information and the SSA representation together provide the information needed for simple or aggressive register allocation algorithms. To support an infinite register set, we use a self-extending instruction encoding, but define a fixed-size 32-bit format to hold small instructions for compactness and translator efficiency. Memory is partitioned into stack, heap, and global memory, and all memory is explicitly allocated. LLVA is a load/store architecture: only `load` and `store` instructions access data values in memory.

LLVA Instructions LLVA has a small, orthogonal instruction set consisting of only the 28 instructions listed in Table 1. The orthogonality makes optimal pattern-matching instruction selectors easier to use. Because almost all instructions are simple, three-address instructions with register operands (`add`, `mul`, `seteq`, etc), the translation process is primarily concerned with combining multiple LLVA instructions into more complex I-ISA instructions wherever possible. Furthermore, the simple low-level operations allow arbitrary machine-independent optimizations to be performed ahead of time by static compilers when generating LLVAcode (unlike JVM bytecode, where operations like array bounds checks, virtual function call resolution, and inlining are difficult to eliminate statically).

LLVA provides low-level operations that can be used to *implement* high-level language features, but in a machine-independent manner. For example, array and structure indexing operations are lowered to typed pointer arithmetic with the `getelementptr` instruction (explained below). Source-level array bounds checks are turned into explicit comparisons. Virtual function dispatch in C++ becomes a pair of `load`s to retrieve the function pointer followed by a `call` (optimizations can eliminate some of these in the static compiler, translator, or both) [26].

Type	Name
arithmetic	add, sub, mul, div, rem
bitwise	and, or, xor, shl, shr
comparison	seteq, setne, setlt, setgt, setle, setge
control-flow	ret, br, mbr, invoke, unwind
memory	load, store, getelementptr, alloca
other	cast, call, phi

Table 1. Entire LLVA Instruction Set

Global Data-flow (SSA) & Control Flow Information A key feature of LLVA that enables efficient dynamic translation is the use of SSA form as the primary representation for scalar register values. SSA form is widely used for compiler optimizations because it allows for efficient "sparse" algorithms for global dataflow problems and provides explicit def-use chains.

To represent SSA information directly in the code, LLVA uses an explicit `phi` instruction to merge values at control-flow join points. (for example, the `%Ret.1` value in Figure 2(b)). The translator elimiantes the ϕ-nodes by introducing copy operations into predecessor basic blocks. These copies are usually eliminated during register allocation.

Exposing an explicit Control flow Graph (CFG) is another crucial feature of LLVA[1]. Each function in LLVA is a list of basic blocks, and each basic block is a list of instructions ending in a single control flow instruction that explicitly specifies its successor basic blocks. A control flow instruction can be a branch, multi-way branch, function return, `invoke` or `unwind`. `invoke` and `unwind` are used to implement source-language exceptions via stack unwind-

1 In contrast, extracting a Control Flow Graph from normal machine code can be quite difficult in practice, due to indirect branches.

```
typedef struct QuadTree {
  double Data;
  struct QuadTree *Children[4];
} QT;

void Sum3rdChildren(QT *T,
                    double *Result) {
  double Ret;
  if (T == 0) { Ret = 0;
  } else {
    QT *Child3 =
      T[0].Children[3];
    double V;
    Sum3rdChildren(Child3, &V);
    Ret = V + T[0].Data;
  }
  *Result = Ret;
}
```

(a) Example function

```
%struct.QuadTree = type { double, [4 x %QT*] }
%QT = type %struct.QuadTree

void %Sum3rdChildren(%QT* %T, double* %Result) {
entry:  %V = alloca double               ;; %V is type 'double*'
        %tmp.0 = seteq %QT* %T, null    ;; type 'bool'
        br bool %tmp.0, label %endif, label %else

else:   ;;tmp.1 = &T[0].Children[3]   'Children' = Field #1
        %tmp.1 = getelementptr %QT* %T, long 0, ubyte 1, long 3
        %Child3 = load %QT** %tmp.1
        call void %Sum3rdChildren(%QT* %Child3, double* %V)
        %tmp.2 = load double* %V
        %tmp.3 = getelementptr %QT* %T, long 0, ubyte 0
        %tmp.4 = load double* %tmp.3
        %Ret.0 = add double %tmp.2, %tmp.4
        br label %endif

endif:  %Ret.1 = phi double [ %Ret.0, %else ], [ 0.0, %entry ]
        store double %Ret.1, double* %Result
        ret void   ;; Return with no value
}
```

(b) Corresponding LLVA code

Figure 2. C and LLVA code for a function

ing, in a manner that is explicit, portable, and can be translated into efficient native code [26].

LLVA Type System The LLVA instruction set is fully-typed, using a low-level, source-language-independent type system. The type system is very simple, consisting of primitive types with predefined sizes (ubyte, uint, float, double, etc...) and 4 derived types (pointer, array, structure, and function). We chose this small set of derived types for two reasons. First, we believe that most high-level language data types are eventually represented using some combination of these low-level types, e.g., a C++ class with base classes and virtual functions is usually represented as a nested structure type with data fields and a pointer to an constant array of function pointers. Second, standard *language-independent* optimizations use only some subset of these types (if any), including optimizations that require array dependence analysis, pointer analysis (even field-sensitive algorithms [16]), and call graph construction.

All instructions in the V-ISA have strict type rules, and most are *overloaded* by type (e.g. '`add int %X, %Y`' vs. '`add float %A, %B`'). There are no mixed-type operations and hence, no implicit type coercion. An explicit `cast` instruction is the sole mechanism to convert a register value from one type to another (e.g. integer to floating point or integer to pointer).

The most important purpose of the type system, however, is to enable typed memory access. LLVA achieves this via type-safe pointer arithmetic using the `getelementptr` instruction. This enables pointer arithmetic to be expressed directly in LLVA without exposing implementation details, such as pointer size or endianness. To do this, offsets are specified in terms of abstract type properties (field number for a structure and element index for an array).

In the example, the `%tmp.1` `getelementptr` instruction calculates the address of `T[0].Children[3]`, by using the symbolic indexes 0, 1, and 3. The "1" index is a result of numbering the fields in the structure. On systems with 32-bit and 64-bit pointers, the offset from the `%T` pointer would be 20 bytes and 32 bytes respectively.

3.2. Representation Portability

As noted in Section 2, a key design goal for a V-ISA is to maintain object code portability across a family of processor implementations. LLVA is broadly aimed to support general-purpose uniprocessors (Section 3.6 discusses some possible extensions). Therefore, it is designed to abstract away implementation details in such processors, including the number and types of registers, pointer size, endianness, stack frame layout, and machine-level calling conventions.

The stack frame layout is abstracted by using an explicit `alloca` instruction to allocate stack space and return a (typed) pointer to it, making all stack operations explicit. As an example, the `V` variable in Figure 2(a) is allocated on the stack (instead of in a virtual register) because its address is taken for passing to `Sum3rdChildren`. In practice, the translator preallocates all fixed-size alloca objects in the function's stack frame at compile time.

The `call` instruction provides a simple abstract calling convention, through the use of virtual register or constant operands. The actual parameter passing and stack adjustment operations are hidden by this abstract, but low-level, instruction.

Pointer size and endianness of a hardware implementation are difficult to completely to abstract away. Type-safe programs which can be compiled to LLVA object code will be automatically portable, without exposing such I-ISA details. Non-type-safe code, however, (e.g., machine-dependent code in C that is conditionally compiled for different platforms) requires exposing such details of the actual I-ISA configuration. For this reason, LLVA includes flags for properties that the source-language compiler can expose to the source program (currently, these are pointer size and endianness). This information is also encoded in the object file so that, using this information, the translator for a different hardware I-ISA can correctly execute the object code (although this emulation would incur a substantial performance penalty on I-ISAs without hardware support).

3.3. Exception Semantics

Previous experience with virtual processor architectures, particularly DAISY and Transmeta, show that there are three especially difficult features to emulate in traditional hardware interfaces: *load/store dependences*, *precise exceptions*, and *self-modifying code*. The LLVA V-ISA already simplifies detecting load/store dependences in one key way: the type, control-flow, and SSA information enable sophisticated alias analysis algorithms in the translator, as discussed in 5.1. For the other two issues also, we have the opportunity to minimize their impact through good V-ISA design.

Precise exceptions are important for implementing many programming languages correctly (without overly complex or inefficient code), but maintaining precise exceptions greatly restricts the ability of compiler optimizations to reorder code. Static compilers often have knowledge about operations that cannot cause exceptions (e.g., a load of a valid global in C), or operations whose exceptions can be ignored for a particular language (e.g., integer overflow in many languages).

We use two simple V-ISA rules to retain precise exceptions but expose non-excepting operations to the translator:

- Each LLVA instruction defines a set of possible exceptions that can be caused by executing that instruction. Any exception delivered to the program is precise, in terms of the visible state of an LLVA program.
- Each LLVA instruction has a boolean attribute named `ExceptionsEnabled`. Exceptions generated by an instruction are ignored if `ExceptionsEnabled` is false for that instruction; otherwise *all* exception conditions are delivered to the program. `ExceptionsEnabled` is *true* by default for `load`, `store` and `div` instructions. It is *false* by default for all other operations, notably all arithmetic operations.

Note also that the `ExceptionsEnabled`attribute is a static attribute and is provided in addition to other mechanisms provided by the V-ABI to disable exceptions dynamically at runtime (e.g. for use in trap handlers).

A second attribute for instructions we are considering would allow exceptions caused by the instruction to be delivered without being precise. Static compilers for languages like C and C++ could flag many untrapped exception conditions (e.g., memory faults) in this manner, allowing the translator to reorder such operations more freely (even if the hardware only supported precise exceptions).

3.4. Self-modifying and Self-extending Code

We use the term Self-Modifying Code (SMC) for a program that explicitly modifies its own pre-existing instructions. We use the term Self-Extending Code (SEC) to refer to programs in which new code is *added* at runtime, but that do not modify any pre-existing code. SEC encompasses several behaviors such as class loading in Java [17], function synthesis in higher-order languages, and program-controlled dynamic code generation. SEC is generally much less problematic for virtual architectures than SMC. Furthermore, most commonly cited examples of "self-modifying code" (e.g., dynamic code generation for very high performance kernels or dynamic code loading in operating systems and virtual machines) are really examples of SEC rather than SMC. Nevertheless, SMC can be useful for implementing runtime code modifications in certain kinds of tools such as runtime instrumentation tools or dynamic optimization systems.

LLVA allows arbitrary SEC, and allows a constrained form of SMC that exploits the execution model for the V-ISA. In particular, a program may modify its own (virtual) instructions via a set of intrinsic functions, but such a change only affects *future invocations* of that function, not any currently active invocations. This ensures that SMC can be implemented efficiently and easily by the translator, simply by marking the function's generated code invalid, forcing it to be regenerated the next time the function is invoked.

3.5. Support for Operating Systems

LLVA uses two key mechanisms to support operating systems and user-space applications: intrinsic functions and a privileged bit. LLVA uses a small set of intrinsic functions to support operations like manipulating page tables and other kernel operations. These intrinsics are implemented by the translator for a particular target. Intrinsics can be defined to be valid only if the privileged bit is set to true, otherwise causing a kernel trap. A trap handler is an ordinary LLVA function with two arguments: the trap number and a pointer of type `void*` to pass in additional information to the handler. Trap handlers can refer to the register state of an LLVA program using a standard, program-independent register numbering scheme for virtual registers. Other intrinsic functions can be used to traverse the program stack and scan stack frames in an I-ISA-independent manner, and to register the entry points for trap handlers.

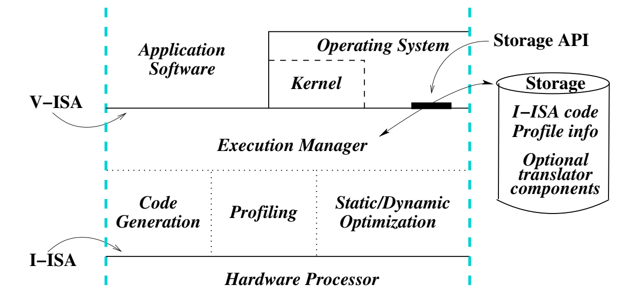

Figure 3. The LLVA execution manager and interface to offline storage.

3.6. Possible Extensions to the V-ISA

Thre are two important kinds of functionality that could be added to the V-ISA. First, the architecture certainly requires definition of synchronization operations and a memory model to support parallel programs. Second, packed operations (also referred to as subword parallelism) are valuable to media and signal-processing codes. These operations must be encoded in the V-ISA because it is difficult for the translator to automatically synthesize them from ordinary sequential code. Finally, we are developing V-ISA extensions that provide *machine-independent abstractions for chip parallelism*. These extensions could be valuable as explicit on-chip parallelism becomes more prevalent (e.g., [33, 21, 31]), raising potentially serious challenges for preserving portability while achieving the highest possible performance across different generations of processors.

4. Translation Strategy

The goals of our translation strategy are to (a) minimize the need for online translation, and (b) to exploit the novel optimization capabilities enabled by a rich, persistent code representation. This paper does not aim to develop new optimization techniques. We are developing such techniques in ongoing research, as part of a complete framework for lifelong code optimization on ordinary processors [26]. Here, we focus on the VISC translation strategy and on the implications of the optimization capabilities for VISC designs.

We begin by describing the "on-chip" runtime execution engine (LLEE) that manages the translation process. We focus in particular on strategies by which it interacts with the surrounding software system to get access to offline storage and enable offline translation. We then describe how the translation strategy exploits the optimization capabilities enabled by a rich persistent code representation.

4.1. LLEE: OS-Independent Translation System

We distinguish two scenarios with different primary constraints on the translation system. The first is when a processor is designed or optimized for a particular OS (e.g., PowerPCs customized for AS/400 systems running IBM's OS/400 [9]). For a VISC processor in such a scenario, the translator can live in offline storage as part of the OS, it can be invoked to perform offline translation, and it can use OS-specific interfaces directly to read and write translations and profile information to offline storage. It can exploit all the optimization mechanisms enabled by the V-ISA, described below. Such a processor should obtain all the benefits of a VISC design without any need for online translation.

More commonly, however, a processor is designed with *no assumptions* about the OS or available storage. The lack of such knowledge places constraints on the translator, as can be seen in DAISY's and Crusoe's translation schemes [11, 14]. Not only is the entire translator program located in ROM, but the translated code and any associated profile information live only in memory and are never cached in persistent storage between executions of a program. Consequently, programs are always translated online after being launched, if the translation does not exist in an in-memory cache.

We propose a translation strategy for such a situation that can enable offline translation and caching, *if an OS ported to LLVA chooses to exploit it*. We have developed a transparent execution environment called LLEE that embodies this strategy, though it is currently implemented at user-level on a standard POSIX system, as described below. It is depicted in Figure 3.

The LLEE translation strategy can be summarized as "offline translation when possible, online translation whenever necessary." A subset (perhaps all) of the translator sufficient for translation and some set of optimizations would live in ROM or flash memory on the processor chip. It is invoked only by LLEE. The V-ABI defines a *standard*, OS-independent storage API with a set of routines that enables LLEE to read, write, and validate data in offline storage. An OS ported to LLVA can choose to implement these routines for higher performance, but they are strictly optional and the system will operate correctly in their absence.

Briefly, the basic storage API includes routines to create, delete, and query the size of an offline cache, read or write a vector of N bytes tagged by a unique string name from/to a cache, and check a timestamp on an LLVA program or on a cached vector. Because these routines are implemented by the OS, and so cannot be linked into the translator, we also define one special LLVA intrinsic routine (recall that an intrinsic is a function implemented by the translator) that the OS can use at startup to register the address of the storage API routine with the translator. This API routine can then be called directly by the translator to query the addresses of other API routines, also at startup. This provides a simple but indefinitely extensible linkage mechanism between translator and OS.

LLEE orchestrates the translation process as follows.

When the OS loads and transfers control to an LLVA executable in memory, LLEE is invoked by the processor hardware. If the OS storage API has been implemented, LLEE uses it to look for a cached translation of the code, checks its timestamp if it exists, and reads it into memory if the translation is not out of date. If successful, LLEE performs relocation as necessary on the native code and then transfers control to it directly. If any condition fails, LLEE invokes the JIT compiler on the entry function. Any new translated code generated by the JIT compiler can be written back to the offline cache if the storage API is available. During idle times, the OS can notify LLEE to perform offline translation of an LLVA program by initiating "execution" as above, but flagging it for translation and not actual execution.

Our implementation of LLEE is faithful to this description except: (a) LLEE is a user-level shared library that is loaded when starting a shell. This library overrides `execve()` with a new version that recognizes LLVA executables and either invokes the JIT on them or executes the cached native translations from the disk, using a user-level version of our storage API. (b) Both the JIT and offline compilers are ordinary programs running on Solaris and Linux, and the offline compiler reads and writes disk files directly. (c) LLVA executables can invoke native libraries not yet compiled to LLVA, e.g., the X11 library.

Finally, the above storage API could be used to allow parts of the translator itself to live offline and to be loaded when needed, minimizing the portion that needs to live in ROM or flash memory. Any component not required for correct execution, such as most aggressive optimizations, could live offline (installed with a little OS cooperation).

4.2. Optimization Strategy

There are important new optimization opportunities created by the rich V-ISA code representation [26], that a VISC architecture can exploit, but most of which are difficult to for programs compiled directly to native code. These include:

1. Compile-time *and link-time* machine-independent optimization (outside the translator).
2. Install-time, I-ISA-specific optimization (before translation).
3. Runtime, trace-driven machine-specific optimization.
4. "Idle-time" (between executions) profile-guided, I-ISA-specific optimization using profile information reflecting actual end-user behavior.

As noted earlier, the LLVA representation allow substantial optimization to be performed before translation, minimizing optimization that must be performed online. Of this, optimization at *link-time* is particularly important because it is the first time that most or all modules of an application are simultaneously available, without requiring changes to application Makefiles and without sacrificing the key benefits of separate compilation. In fact, many commercial compilers today perform interprocedural optimization at link-time, by exporting their proprietary compiler internal representation during static compilation [3, 20]. Such compiler-specific solutions are unnecessary with LLVA because it retains rich enough information to support extensive optimizations, as demonstrated in 5.1.

Install-time optimization is just an application of the translator's optimization and code generation capabilities to generate carefully tuned code for a particular system configuration. This is a direct benefit of retaining a rich code representation until software is installed, while still retaining the ability to do offline code generation.

Unlike other trace-driven runtime optimizers for native binary code, such as Dynamo [4], we have both the rich V-ISA and a cooperating code generator. Our V-ISA provides us with ability to perform static instrumentation to assist runtime path profiling, and to use the CFG *at runtime* to perform path profiling within frequently executed loop regions while avoiding interpretation. It also lets us develop an aggressive optimization strategy that operates on traces of LLVA code corresponding to the hot traces of native code. We have implemented the tracing strategy and software trace cache, including the ability to gather cross-procedure traces, [26], and we are now developing runtime optimizations that exploit these traces.

The rich information in LLVA also enables "idle-time" profile-guided optimization (PGO) using the translator's optimization and code generation capabilities. The important advantage is that this step can use profile information gathered from executions on an end-user's system. This has three distinct advantages over static PGO: (a) the profile information is more likely to reflect end-user behavior than hypothetical profile information generated by developers using predicted input sets; (b) developers often do not use profile-guided optimization or do so only in limited ways, whereas "idle-time" optimization can be completely transparent to users, if combined with low-overhead profiling techniques; and (c) idle-time optimization can combine profile information with detailed information about the user's specific system configuration.

5. Initial Evaluation

We believe the performance implications of a Virtual ISA design cannot be evaluated meaningfully without (at least) a processor design with hardware mechanisms that support translation and optimization [11]), and (preferably) basic cooperative hardware/software mechanisms that exploit the design. Since the key contribution of this paper is the design of LLVA, we focus on evaluating the features of this design. In particular, we consider the 2 questions listed in the Introduction: does the representation enable high-level analysis and optimizations, and is the representation low-level enough to closely match with hardware and to be translated efficiently?

5.1. Supporting High Level Optimizations

The LLVA code representation presented in this paper is also used as the internal representation of a sophisticated compiler framework we call Low Level Virtual Machine (LLVM) [26]. LLVM includes front-ends for C and C++ based on GCC, code generators for both Intel IA-32 and SPARC V9 (each can be run either offline or as a JIT compiling functions on demand), a sophisticated link-time optimization system, and a software trace cache. Compared with the instruction set in Section 3, the differences in the compiler IR are: (a) the compiler extracts type information for memory allocation operations and converts them into typed `malloc` and `free` instructions (the back-ends translate these back into the library calls), and (b) the `ExceptionEnabled` bit is hardcoded based on instruction opcode. The compiler system uses equivalent internal and external representations, avoiding the need for complex translations at each stage of the compilation process.

The compiler uses the virtual instruction set for a variety of analyses and optimizations including many classical dataflow and control-flow optimizations, as well as more aggressive link-time interprocedural analyses and transformations. The classical optimizations directly exploit the control-flow graph, SSA representation, and several choices of pointer analysis. They are usually performed on a per module-basis, before linking the different LLVA object code modules, but can be performed at any stage of a program's lifetime where LLVA code is available.

We also perform several novel interprocedural techniques using the LLVA representation, all of which operate at link-time. *Data Structure Analysis* is an efficient, context-sensitive pointer analysis, which computes both an accurate call graph and points-to information. Most importantly, it is able to identify *information about logical data structures* (e.g., an entire list, hashtable, or graph), including disjoint instances of such structures, their lifetimes, their internal static structure, and external references to them. *Automatic Pool Allocation* is a powerful interprocedural transformation that uses Data Structure Analysis to partition the heap into separate pools for each data structure instance [25]. Finally, we have shown that the LLVA representation is rich enough to perform *complete, static analysis* of memory safety for a large class of type-safe C programs [24, 13]. This work uses both the techniques above, plus an interprocedural array bounds check removal algorithm [24] and some custom interprocedural dataflow and control flow analyses [13].

The interprocedural techniques listed above are traditionally considered very difficult even on source-level imperative languages, and are impractical for machine code. In fact, all of these techniques fundamentally require type information for pointers, arrays, structures and functions in LLVA plus the Control Flow Graph. The SSA representation significantly improves both the precision and speed of the analyses and transformations. Overall, these examples amply demonstrate that the virtual ISA is rich enough to support powerful (language-independent) compiler tasks traditionally performed only in source-level compilers.

5.2. Low-level Nature of the Instruction Set

Table 2 presents metrics to evaluate the low-level nature of the LLVA V-ISA. The benchmarks we use include the PtrDist benchmarks [2] and the SPEC CINT2000 benchmarks (we omit three SPEC codes because their LLVAobject code versions fail to link currently). The first two columns in the table list the benchmark names and the number of lines of C source code for each.

Columns 3 and 4 in the table show the fully linked code sizes for a statically compiled native executable and for the LLVA object program. The native code is generated from the LLVA object program using our static back end for SPARC V9. These numbers are comparable because the same LLVA optimizations were applied in both cases. The numbers show that the virtual object code is significantly smaller than the native code, roughly 1.3x to 2x for the larger programs in the table (the smaller programs have even larger ratios)[2]. Overall, despite containing extra type and control flow information and using SSA form, the virtual code is still quite compact for two reasons. First, most instructions usually fit in a single 32-bit word. Second, the virtual code does not include verbose machine-specific code for argument passing, register saves and restores, loading large immediate constants, etc.

The next five columns show the number of LLVA instructions, the total number of machine instructions generated by the X86 back-end, and the ratio of the latter to the former (and similarly for Sparc). The x86 back-end performs virtually no optimization and very simple register allocation resulting in significant spill code. The Sparc back-end produces higher quality code, but requires more instructions because of the RISC architecture. Nevertheless, each LLVA instruction translates into very few I-ISA instructions on average; about 2-3 for X86 and 2.5-4 for SPARC V9. Furthermore, *all* LLVA instructions are translated directly to native machine code – no emulation routines are used at all. These results indicate that the LLVA instruction set uses *low-level operations that match closely with native hardware instructions*.

Finally, the last three columns in the table show the total code generation time taken by the X86 JIT compiler to

2 The GCC compiler generates more compact SPARC V8 code, which is roughly equal in size to the bytecode [26].

Program	#LOC	Native size (KB)	LLVAcode size (KB)	#LLVA Inst.	#X86 Inst.	Ratio	#SPARC Inst.	Ratio	Translate Time (s)	Run time (s)	Ratio
ptrdist-anagram	647	21.7	10.7	776	1817	2.34	2550	3.29	0.0078	1.317	0.006
ptrdist-ks	782	24.9	12.1	1059	2732	2.58	4446	4.20	0.0039	1.694	0.002
ptrdist-ft	1803	20.9	10.1	799	1990	2.49	2818	3.53	0.0117	2.797	0.004
ptrdist-yacr2	3982	58.3	36.5	4279	10881	2.54	12252	2.86	0.0429	2.686	0.016
ptrdist-bc	7297	112.0	74.4	7276	19286	2.65	25697	3.53	0.1308	1.307	0.100
179.art	1283	37.8	17.9	2027	5385	2.66	7031	3.47	0.0253	114.723	0.000
183.equake	1513	44.4	23.9	2863	6409	3.14	8275	2.89	0.0273	18.005	0.002
181.mcf	2412	32.0	17.3	2039	4707	2.31	4601	2.26	0.0175	24.516	0.001
256.bzip2	4647	73.5	55.7	5103	11984	2.35	14157	2.77	0.0371	20.896	0.002
164.gzip	8616	94.0	68.6	7594	17500	2.30	20880	2.75	0.0527	19.332	0.003
197.parser	11391	223.0	175.3	17138	41671	2.43	57274	3.34	0.1601	4.718	0.034
188.ammp	13483	265.1	163.2	21961	53529	2.44	67679	3.08	0.1074	58.758	0.002
175.vpr	17729	331.0	184.4	18041	58982	3.27	74696	4.14	0.1425	7.924	0.018
300.twolf	20459	487.7	330.0	45017	104613	2.32	119691	2.66	0.0156	9.680	0.002
186.crafty	20650	555.5	336.4	34080	104093	3.05	110630	3.25	0.4531	15.408	0.029
255.vortex	67223	976.3	719.3	72039	195648	2.72	224488	3.12	0.7773	6.753	0.115
254.gap	71363	1088.1	854.4	111482	246102	2.21	272483	2.44	0.4824	3.729	0.129

Table 2. Metrics demonstrating code size and low-level nature of the V-ISA

compile the entire program (regardless of which functions are actually executed), the total running time of each program when compiled natively for X86 using `gcc -O3`, and the ratio of the two. As the table shows, the JIT compilation times are negligible, except for large codes with short running time. Furthermore, this behavior should extend to much larger programs as well because the JIT translates functions on demand, so that unused code is not translated (we show the compilation time for the entire program, since that makes the data easier to understand). Overall, this data shows that it is possible to do a very fast, non-optimizing translation of LLVA code to machine code at very low cost. Any support to translate code offline and/or to cache translated code offline should further reduce the impact of this translation cost.

Overall, both the instruction count ratio and the JIT compilation times show that the LLVA V-ISA is very closely matched to hardware instruction sets in terms of the complexity of the operations, while the previous subsection showed that it includes enough high-level information for sophisticated compiler optimizations. This combination of high-level information with low-level operations is the crucial feature that (we believe) makes the LLVA instruction set a good design for a Virtual Instruction Set Architecture.

6. Related Work

Virtual machines of different kinds have been widely used in many software systems, including operating systems (OS), language implementations, and OS and hardware emulators. These uses do not define a Virtual ISA at the hardware level, and therefore do not directly benefit processor design (though they may influence it). The challenges of using two important examples – Java Virtual Machine and Microsoft CLI – as a processor-level virtual ISA were discussed in the Introduction.

We know of four previous examples of VISC architectures, as defined in Section 1: the IBM System/38 and AS/400 family [9], the DAISY project at IBM Research [14], Smith et al.'s proposal for Codesigned Virtual Machines in the Strata project [32], and Transmeta's Crusoe family of processors [23, 11]. All of these distinguish the virtual and physical ISAs as a fundamental processor design technique. To our knowledge, however, *none except the IBM S/38 and AS/400 have designed a virtual instruction set for use in such architectures*.

The IBM AS/400, building on early ideas in the S/38, defined a Machine Interface (MI) that was very high-level, abstract and hardware-independent (e.g., it had no registers or storage locations). It was the sole interface for all application software and for much of OS/400. Their design, however, differed from ours in fundamental ways, and hence does not meet the goals we laid out in Section 2. Their MI was targeted at a particular operating system (the OS/400), it was designed to be implemented using complex operating system and database services and not just a translator, and was designed to best support a particular workload class, viz., commercial database-driven workloads. It also had a far more complex instruction set than ours (or any CISC processors), including string manipulation operations, and "object" manipulation operations for 15 classes of objects (e.g., programs and files). In contrast, our V-ISA is philosophically closer to modern processor instruction sets in being a minimal, orthogonal, load/store architecture; it is OS-independent and requires no software other than a translator; and it is designed to support modern static and dynamic optimization techniques for general-purpose software.

DAISY [14] developed a dynamic translation scheme for emulating multiple existing hardware instruction sets (PowerPC, Intel IA-32, and S/390) on a VLIW processor. They developed a novel translation scheme with global VLIW scheduling fast enough for online use, and hardware extensions to assist the translation. Their translator operated on a page granularity. Both the DAISY and Transmeta translators are stored entirely in ROM on-chip. Because they focus on existing V-ISAs with existing OS/hardware interface specifications, they cannot assume any OS support and thus cannot cache any translated code or profile information in off-processor storage, or perform any offline translation.

Transmeta's Crusoe uses a dynamic translation scheme to emulate Intel IA-32 instructions on a VLIW hardware processor [23]. The hardware includes important supporting mechanisms such as shadowed registers and a gated store buffer for speculation and rollback recovery on exceptions, and alias detection hardware in the load/store pipeline. Their translator, called Code Morphing Software (CMS), exploits these hardware mechanisms to reorder instructions aggressively in the presence of the challenging features identified in Section 3.3, namely, precise exceptions, memory dependences, and self-modifying code (as well as memory-mapped I/O) [11]. They use a trace-driven reoptimization scheme to optimize frequently executed dynamic sequences of code. Crusoe does do not perform any offline translation or offline caching, as noted above.

Smith et al. in the Strata project have recently but perhaps most clearly articulated the potential benefits of VISC processor designs, particularly the benefits of co-designing the translator and a hardware processor with an implementation-dependent ISA [32]. They describe a number of examples illustrating the flexibility hardware designers could derive from this strategy. They have also developed several hardware mechanisms that could be valuable for implementing such architectures, including relational profiling [19], a microarchitecture with a hierarchical register file for instruction-level distributed processing [22], and hardware support for working set analysis [12]. They do not propose a specific choice of V-ISA, but suggest that one choice would be to use Java VM as the V-ISA (an option we discussed in the Introduction).

Previous authors have developed Typed Assembly Languages [28, 7] with goals that generally differ significantly from ours. Their goals are to enable compilation from strongly typed high-level languages to typed assembly language, enabling sound (type preserving) program transformations, and to support program safety checking. Their type systems are higher-level than ours, because they attempt to propagate significant type information from source programs. In comparison, our V-ISA uses a much simpler, low-level type system aimed at capturing the common low-level representations and operations used to implement computations from high-level languages. It is also designed to to support arbitrary non-type-safe code efficiently, including operating system and kernel code.

Binary translation has been widely used to provide binary compatibility for legacy code. For example, the FX!32 tool uses a combination of online interpretation and offline profile-guided translation to execute Intel IA-32 code on Alpha processors [8]. Unlike such systems, a VISC architecture makes binary translation *an essential part of the design strategy*, using it for all codes, not just legacy codes.

There is a wide range of work on software and hardware techniques for transparent dynamic optimization of programs. Transmeta's CMS [11] and Dynamo [4] identify and optimize hot traces at runtime, similar to our reoptimization strategy but without the benefits of a rich V-ISA. Many JIT compilers for Java, Self, and other languages combine fast initial compilation with adaptive reoptimization of "hot" methods (e.g., see [1, 6, 18, 34]). Finally, many hardware techniques have been proposed for improving the effectiveness of dynamic optimization [27, 30, 35]. When combined with a rich V-ISA that supports more effective program analyses and transformations, these software and hardware techniques can further enhance the benefits of VISC architectures.

7. Conclusions and Future Work

Trends in modern processors indicate that CPU cycles and raw transistors are becoming increasingly cheap, while control complexity, wire delays, power, reliability, and testing cost are becoming increasingly difficult to manage. *Both trends favor virtual processor architectures*: the extra CPU cycles can be spent on software translation, the extra transistors can be spent on mechanisms to assist that translation, and a cooperative hardware/software design supported by a rich virtual program representation could be used in numerous ways to reduce hardware complexity and potentially increase overall performance.

This paper presented LLVA, a design for a language-independent, target-independent virtual ISA. The instruction set is low-level enough to map directly and closely to hardware operations but includes high-level type, control-flow and dataflow information needed to support sophisticated analysis and optimization. It includes novel mechanisms to overcome the difficulties faced by previous virtual architectures such as DAISY and Transmeta's Crusoe, including a flexible exception model, minor constraints on self-modifying code to dovetail with the compilation strategy, and an OS-independent interface to access offline storage and enable offline translation.

Evaluating the benefits of LLVA requires a long-term research program. We have three main goals in the near future: (a) Develop and evaluate cooperative (i.e., codesigned) software/hardware design choices that reduce hardware complexity and assist the translator to achieve high overall performance. (b) Extend the V-ISA with machine-independent abstractions of fine- and medium-grain parallelism, suitable for mapping to explicitly parallel processor designs, as mentioned in Section 3.6. (c) Port an existing operating system (in incremental steps) to work on top of the LLVA architecture, and explore the OS design implications of such an implementation.

Acknowledgements

We thank Jim Smith, Sarita Adve, John Criswell and the anonymous referees for their detailed feedback on this paper. This work has been supported by an NSF CAREER award, EIA-0093426, the NSF Operating Systems and Compilers program under grant number CCR-9988482, and the SIA's MARCO Focus Center program.

References

[1] A.-R. Adl-Tabatabai, et al. Fast and effective code generation in a just-in-time Java compiler. In *PLDI*, May 1998.

[2] T. Austin, et al. The pointer-intensive benchmark suite. Available at `www.cs.wisc.edu/~austin/ptr-dist.html`, Sept 1995.

[3] A. Ayers, S. de Jong, J. Peyton, and R. Schooler. Scalable cross-module optimization. *ACM SIGPLAN Notices*, 33(5):301–312, 1998.

[4] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *PLDI*, pages 1–12, June 2000.

[5] D. Burger and J. R. Goodman. Billion-transistor architectures. *Computer*, 30(9):46–49, Sept 1997.

[6] M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *Java Grande*, pages 129–141, 1999.

[7] J. Chen, D. Wu, A. W. Appel, and H. Fang. A provably sound TAL for back-end optimization. In *PLDI*, San Diego, CA, Jun 2003.

[8] A. Chernoff, et al. FX!32: A profile-directed binary translator. *IEEE Micro*, 18(2):56–64, 1998.

[9] B. E. Clark and M. J. Corrigan. Application system/400 performance characteristics. *IBM Systems Journal*, 28(3):407–423, 1989.

[10] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, pages 13(4):451–490, October 1991.

[11] J. C. Dehnert, et al. The Transmeta Code Morphing Software: Using speculation, recovery and adaptive retranslation to address real-life challenges. In *Proc. 1st IEEE/ACM Symp. Code Generation and Optimization*, San Francisco, CA, Mar 2003.

[12] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA*, Alaska, May 2002.

[13] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner. Memory safety without runtime checks or garbage collection. In *LCTES*, San Diego, CA, Jun 2003.

[14] K. Ebcioglu and E. R. Altman. DAISY: Dynamic compilation for 100% architectural compatibility. In *ISCA*, pages 26–37, 1997.

[15] J. Fisher. Walk-time techniques: Catalyst for architectural change. *Computer*, 30(9):46–42, Sept 1997.

[16] R. Ghiya, D. Lavery, and D. Sehr. On the importance of points-to analysis and other memory disambiguation methods for C programs. In *PLDI*. ACM Press, 2001.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 2nd Ed.* Addison-Wesley, Reading, MA, 2000.

[18] D. Griswold. The Java HotSpot Virtual Machine Architecture, 1998.

[19] T. H. Heil and J. E. Smith. Relational profiling: enabling thread-level parallelism in virtual machines. In *MICRO*, pages 281–290, Monterey, CA, Dec 2000.

[20] IBM Corp. XL FORTRAN: Eight Ways to Boost Performance. White Paper, 2000.

[21] Intel Corp. Special Issue on Intel HyperThreading Technology in Pentium 4 Processors. *Intel Technology Journal*, Q1, 2002.

[22] H.-S. Kim and J. E. Smith. An instruction set and microarchitecture for instruction level distributed processing. In *ISCA*, Alaska, May 2002.

[23] A. Klaiber. The Technology Behind Crusoe Processors, 2000.

[24] S. Kowshik, D. Dhurjati, and V. Adve. Ensuring code safety without runtime checks for real-time control systems. In *CASES*, Grenoble, France, Oct 2002.

[25] C. Lattner and V. Adve. Automatic Pool Allocation for Disjoint Data Structures. In *Proc. ACM SIGPLAN Workshop on Memory System Performance*, Berlin, Germany, Jun 2002.

[26] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. Tech. Report UIUCDCS-R-2003-2380, Computer Science Dept., Univ. of Illinois at Urbana-Champaign, Sept 2003.

[27] M. C. Merten, A. R. Trick, E. M. Nystrom, R. D. Barnes, and W. m. W. Hwu. A hardware mechanism for dynamic extraction and relayout of program hot spots. In *ISCA*, pages 59–70, Jun 2000.

[28] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *TOPLAS*, 21(3):528–569, May 1999.

[29] P. Oberoi and G. S. Sohi. Parallelism in the front-end. In *ISCA*, June 2003.

[30] S. J. Patel and S. S. Lumetta. rePLay: A Hardware Framework for Dynamic Optimization. *IEEE Transactions on Computers*, Jun 2001.

[31] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, , and J. Huh. Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture. In *ISCA*, June 2003.

[32] J. E. Smith, T. Heil, S. Sastry, and T. Bezenek. Achieving high performance via co-designed virtual machines. In *International Workshop on Innovative Architecture (IWIA)*, 1999.

[33] J. M. Tendler, J. S. Dodson, J. S. Fields, Jr., H. Le, and B. Sinharoy. The POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, 2002.

[34] D. Ungar and R. B. Smith. Self: The power of simplicity. In *OOPSLA*, 1987.

[35] C. Zilles and G. Sohi. A programmable coprocessor for profiling. In *HPCA*, Jan 2001.