# ISAMAP: Instruction Mapping Driven by Dynamic Binary Translation

Maxwell Souza, Daniel Nicácio, Guido Araújo

# ISAMAP: Instruction Mapping Driven by Dynamic Binary Translation

Maxwell Souza
Institute of Computing
UNICAMP, Brazil
maxwell.monteiro@gmail.com

Daniel Nicácio
Institute of Computing
UNICAMP, Brazil
dnicacio@ic.unicamp.br

Guido Araújo
Institute of Computing
UNICAMP, Brazil
guido@ic.unicamp.br

***Abstract*—Dynamic Binary Translation (DBT) techniques have been largely used in the migration of legacy code and in the transparent execution of programs across different architectures. They have also been used in dynamic optimizing compilers, to collect runtime information so as to improve code quality. In many cases, DBT translation mechanism misses important low-level mapping opportunities available at the source/target ISAs. Hot code performance has been shown to be central to the overall program performance, as different instruction mappings can account for high performance gains. Hence, DBT techniques that provide efficient instruction mapping at the ISA level has the potential to considerably improve performance. This paper proposes ISAMAP, a flexible instruction mapping driven by dynamic binary translation. Its mapping mechanism, provides a fast translation between ISAs, under an easy-to-use description. At its current state, ISAMAP is capable of translating 32-bit PowerPC code to 32-bit x86 and to perform local optimizations on the resulting x86 code. Our experimental results show that ISAMAP is capable of executing PowerPC code on an x86 host faster than the processor emulator QEMU, achieving speedups of up to 3.16x for SPEC CPU2000 programs.**

## I. Introduction

With the availability of 64-bit multi-core processor instructions sets like EM64T [1], AMD64 [2] and others (e.g. PowerPC), new cross platform optimization opportunities have surfaced. Although old 32-bit programs can still run on such architectures, they do not take full advantage of some 64-bit new architecture features (i.e. the additional number of registers). Moreover, consumer electronics have been opening up new opportunities for low-power general-purpose processors, which eventualy could benefit from DBT by running programs not compiled for that domain.

Running code in a DBT environment can considerably impact the program execution time, due to the time required to translate instructions into the new target ISA. On multi-core processors this overhead can be considerably reduced, by assigning one thread to do binary translation, optimization and profiling, while another thread executes the translated code [1] [2].

[1] *EM64T (Extended Memory 64-bit Technology)* Intel's implementation of x86-64 architecture. It is used in newer versions of Pentium 4, Pentium D, Pentium Extreme Edition, Celeron D, Xeon, and Pentium Dual-Core processors, and in all versions of the Core 2 processors.

[2] AMD64 (aka x86-64) is a 64-bit superset of the x86 instruction set architecture developed by AMD.

This paper presents ISAMAP, an instruction mapping driven by dynamic binary translator, which uses dynamic ISA mapping in code translation. Source program instructions are decoded, mapped and encoded into host code by following the instruction set descriptions of the involved architectures. These descriptions, in assembly-like language, allows for an efficient mapping as it taps on the low-level machine code features of each architecture. The result is a good quality code for the target architecture. In the specific case of the PowerPC to x86 translation, ISAMAP achieves speedups of up to 3.16x when comparing against QEMU running SPEC CPU2000 programs.

This paper is divided as follows. Section II presents some popular Dynamic Binary Translators (DBTs). Section III shows the ISAMAP environment, how it works, format of the description, mapping statements and some issues related to the binary translation process. Section IV shows experimental results on dynamic translation between PowerPC to x86 code and optimization speedups. Section V concludes with some observations and future work.

## II. Related Work

Aries [3] does binary translation between PA-RISC and IA-64 code using a fast interpretation mechanism. The interpreter counts how many times each block is interpreted, once the block counter reaches a threshold it is translated into native code. Translated blocks are stored in code cache and restored when necessary. System calls, signals and exceptions are handled by the Environment Emulation System.

Digital FX!32 [4] executes x86 code on an Alpha System by employing emulation and static binary translation. The x86 program is totally interpreted during the first time it executes and information about its execution behavior is stored (profiling). Once the interpreted execution is completed, FX!32 system performs program translation based on the data collected during the profiling execution. Translated code is stored in a database. Next time the program is requested to run, FX!32 collects the translated code from a database and executes it.

IA-32 EL [5] allows IA-32 code execution on Itanium based systems through interpretation and dynamic binary translation. IA-32 EL uses two mechanisms: Cold Code Translation, which interprets IA-32 code and Hot Code Translation, which translates the most executed blocks to form a super-block or a trace.

Translated blocks are stored into code cache and retrieved every time it is needed. Every exception thrown is redirected to the IA-32 EL system, which translates the exception to an IA-32 exception before passing it to the original application. In integer benchmarks, IA-32 EL executes translated code with 65% of the original code performance.

Dynamo [6] translates PA-8000 code to PA-8000. The PA-8000 code is interpreted until a hot region is found (traces, loops, etc). Once a hot region is found, optimizations are performed on code and it is stored into a code cache. Optimizations performed by Dynamo achieve speedups up to 1.22x in some benchmarks.

In UQDBT [7] a set of specifications defines binary code format, instructions syntax and semantic, allowing a very flexible system. UQDBT is also capable of identifying hot paths and apply optimizations. UQDBT has slowdown between 2.5x and 7.0x when compared to native code. When optimizations are turned on, slowdown is reduced in 15%.

ADORE [8] is a dynamic binary optimization framework with hardware (Itanium 2) mechanisms to help the code analysis during execution. The ADORE system tries to identify the most frequently executed traces, which are built with the help of dedicated registers that store the last four taken branches. Once the trace is built, several optimizations can be applied: register allocation, data cache pre-fetching and trace layout. Due to its hardware support, ADORE has a low overhead, 2%, and can achieve speedups up to 2.56x.

Yirr-Ma [9] is another dynamic binary translation framework that uses customized code instrumentation. Like other DBTs, it collects code behavior information through emulation and uses it to translate and optimize binary code. Yirr-Ma uses the WalkAbout framework, a framework that automatically generates emulators from SLED and SSL specifications. Since Yirr-Ma is an emulator, each source ISA instruction is represented as a function in a high-level language (eg. C). This function carries instrumentation code that allows identifying hot regions on the emulated code. Once hot regions are known, they can be translated to host code and optimized.

DAISY [10] is a dynamic emulator designed to target VLIW architectures. It is capable of transalating PowerPC code to VLIW primitives, then it optimizes the resulting code to take full advantage of the target architecture. The final code shows good instruction level paralelism.

QEMU [11] is a fast open source emulator that uses binary translation and runs several different source ISAs on many target hosts, including x86, x86_64 and PowerPC. The instruction mapping is performed by using C functions like in Yirr-MA [9]. These functions are compiled and its object code linked with the emulator code. The encoding process is done by a simply copy and paste method that allows a very fast host code encoding. Code cache and block linkage mechanisms guarantee a great performance, considering QEMU is an emulator. QEMU executes PowerPC code on x86 host between 4 and 10 times slower than native x86 code.

ISAMAP holds some similarities and differences when compared to the above mentioned systems. Unlike ISAMAP, DYNAMO and ADORE use dynamic translation to optimize code on the fly, and thus need no instruction mapping. The remaining systems translate code between different architectures. While most of them use a generic intermediate language, free of architecture restrictions, IA-32 EL and ISAMAP are similar in the sense that both use a direct instruction mapping mechanism. As far as we know, IA-32 EL and ISAMAP are both capable of adaptative translation, based on the instruction parameters. On the other hand, ISAMAP uses a very simple syntax, similar to those found in well-known compiler front-end and code-generator generation tools [12], [13]. As the experiments reveal, this exposes low-level mapping opportunities which result in an afficient way to translate code between architectures.

## III. ISAMAP

ISAMAP descriptions are a subset of ArchC [14], an architecture description language, which has been largely used to synthesize processor simulators and assemblers [15]. By using ArchC, it is possible to describe both functional and cycle accurate processor models. ISAMAP requires three ArchC models, one for the source processor, another for the target ISA and one to describe the instruction mapping between them. By using ArchC, ISAMAP can easily capture processor instructions format, registers, operands, and instruction mapping. Figure 1 shows an example of PowerPC ISA description model and figure 2 shows a x86 ISA description model. From these models, part of the binary translator source code is automatically generated.

```
ISA(powerpc) {
  isa_format XO1 = "%opcd:6 %rt:5 %ra:5 %rb:5 %oe:1
                    %xos:9 %rc:1";
  isa_instr <XO1> add, subf;
  isa_regbank r:32 = [0..31];
  ISA_CTOR(powerpc) {
    add.set_operands("%reg %reg %reg", rt, ra, rb);
    add.set_decoder(opcd=31, oe=0, xos=266, rc=0);
    subf.set_operands("%reg %reg %reg", rt, ra, rb);
    subf.set_decoder(opcd=31, oe=0, xos=40, rc=0);
  }
}
```

Figure 1. PowerPC ISA description.

```
1  ISA(x86) {
2    isa_format op1b_r32 = "%op1b:8 %mod:2 %regop:3 %rm:3";
3    isa_instr <op1b_r32> add_r32_r32, mov_r32_r32;
4    isa_reg eax = 0;
5    isa_reg ecx = 1;
6    ...
7    isa_reg edi = 7;
8    ISA_CTOR(x86) {
9      add_r32_r32.set_operands("%reg %reg", rm, regop);
10     add_r32_r32.set_encoder(op1b=0x01, mod=0x3);
11     mov_r32_r32.set_operands("%reg %reg", rm, regop);
12     mov_r32_r32.set_encoder(op1b=0x89, mod=0x3);
13   }
14 }
```

Figure 2. X86 ISA description.

### A. Models

The main ISAMAP fields used to describe the source/target ISA and the mapping between them are listed below. ISAMAP uses such fields to automatically synthesize the source/target ISA decoder/encoder.

- `isa_format`: declares an instruction format, its fields and size in bits;
- `isa_instr`: instantiates instructions and assign them to their respective formats;
- `isa_reg`: defines names and opcodes for the processor registers;
- `isa_regbank`: defines register banks and the register interval for bank registers;
- `set_operands`: specifies instruction operands, their types and to which field they are assigned to. There are three possible operand types: `reg`: register, `addr`: address and `imm`: immediate.
- `set_encoder` and `set_decoder`: describes which fields identify an instruction and their respective values.

Figure 2 shows an example on how to use the ISAMAP fields. Line 2 `isa_format` defines a format named (`op1b_r32`) containing four fields of sizes: 8, 2, 3, 3 bits. Line 3 `isa_instr` declares two instructions (`add_r32_r32` and `mov_r32_r32`), which belong to format `op1b_r32`. From line 4 to 7, registers are declared and their opcode is defined by the keyword `isa_reg`. Lines 9 and 11 define operands of instructions `add_r32_r32` and `mov_r32_r32`. In the two instructions, field `rm` is the first operand and field `regop` the second. Lines 10 and 12 define values for the instruction operands, but in this case, only fields `op1b` and `mod` have their values declared.

The instruction mapping between source and target ISA is described in a third description. For each source architecture instruction, its behavior is mapped into one or more instructions. As shown in figure 3, the syntax of the description is similar to the one used in code-generator tools like iburg [13] and twig [12].

```
isa_map_instrs {
  add %reg %reg %reg;
} = {
  mov_r32_r32 edi $1;
  add_r32_r32 edi $2;
  mov_r32_r32 $0 edi;
}
```

Figure 3. Mapping sample PowerPC to x86.

Although this approach is not as portable as the mapping scheme used by Yirr-MA [9], it allows the generation of faster code and provides enough flexibility to implement different mappings for each source architecture instruction. As an example, the mapping shown in figure 3 could also be described by an `lea` instruction, thus resulting in one less instruction mapping. ISAMAP leverages on similar opportunities to considerably improve the quality of the target processor code.

Symbols like *edi* are used when a specific register of the target architecture is required, in this case, register *edi*. Symbols started by $ character indicate a reference to an instruction operand in the source architecture instruction, $0 refer to operand 0, $1 to operand 1 and so on. If the referenced operand is a register, the register in the target architecture, which maps the source architecture register, is defined in the instruction. Figure 4 shows an example of the resulting code after mapping the PowerPC instruction `add r0, r1, r3` to x86 code. Instructions in lines 0, 2, 5 are spill code, and will be addressed later.

```
0 mov eax, 0x80740504
1 mov edi, eax
2 mov eax, 0x80740508
3 add edi, eax
4 mov eax, edi
5 mov 0x80740500, eax
```

Figure 4. Generated code sample.

The instruction in line 0 loads the content of register R1, which is mapped to memory, as EAX. In line 2, register R3 is loaded into EAX. In line 5, the operation result is stored into register R0. This code sample can be tuned up if another `add` instruction mapping is used. Notice that the x86 architecture allows instruction operands to be a memory reference (but not all of them). If that is the case, instructions `add` and `mov` are used, resulting in a generated code with at least three fewer instructions. Figure 5 shows the alternative specification of instructions `add` and `mov`, when having one operand as a memory reference. Figure 6 illustrates the new mapping and figure 7 shows the generated code for the new mapping. As it can be seen, the new mapping has only three instructions.

```
...
isa_format op1b_r32_m32disp = "%op1b:8 %mod:2 %regop:3
                               %rm:3 %m32disp:32";
isa_instr <op1b_r32_m32disp> add_r32_m32disp,
                             mov_r32_m32disp;
...
add_r32_m32disp.set_operands("%reg %addr", regop, m32disp);
add_r32_m32disp.set_encoder(op1b=0x01, mod=0x0, rm=0x5);
mov_r32_m32disp.set_operands("%reg %addr", regop, m32disp);
mov_r32_m32disp.set_encoder(op1b=0x8b, mod=0x0, rm=0x5);
mov_m32disp_r32.set_operands("%addr %reg", m32disp, regop);
mov_m32disp_r32.set_encoder(op1b=0x89, mod=0x0, rm=0x5);
```

Figure 5. Another example of x86 instructions specification.

```
isa_map_instrs {
  add %reg %reg %reg;
} = {
  mov_r32_m32disp edi $1;
  add_r32_m32disp edi $2;
  mov_m32disp_r32 $0 edi;
}
```

Figure 6. The new PowerPC to x86 mapping.

```
0 mov edi, 0x80740504
1 add edi, 0x80740508
2 mov 0x80740500, edi
```

Figure 7. The new generated code.

### B. System Overview

The structure of binary translator ISAMAP follows the same structure of most binary translators available in the literature. Figure 8 illustrates ISAMAP overall structure.

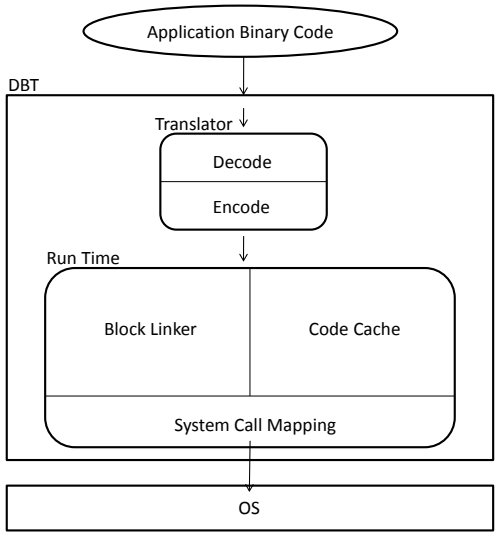


Figure 8. ISAMAP structure

The translation module is formed by two sub-modules: decode and encode. The Run-Time module contains the following sub-modules: Code Cache, Block Linker and System Call Mapping. Code Cache stores basic blocks already translated, thus allowing fast target code retrieval when needed. Block Linker, as the names states, links basic blocks, thus avoiding the intervention of Run-Time module every time a basic block is executed. At last, the System Call Mapping module is responsible for the mapping, during the translation process, between the system calls with different implementations in the involved architectures.

### C. Translator Generation

The Translator Generator receives as input the source, target, and mapping descriptions and then generates the translator's source code in C, `translator.c`. Code in `translator.c` is responsible for making calls to the decoder and a C-switch block makes the mapping and code emission (into target architecture instructions). This process is performed for each source architecture decoded instruction. The following files are also generated:

- *ctx_switch.c*: Responsible for the emission of binary code that stores and loads the target architecture registers content;
- *isa_init.c*: Initializes the data structures that hold information about instructions, formats and field of the source architecture;
- *encode_init.c*: Initializes data structures that hold information about instructions, formats and field of the target architecture;
- *pc_update.c*: Defines function prototypes, which are responsible for emulating branch instructions. Implementation of this function must be provided.
- *spill.c*: Defines function prototypes for spill code emission. Implementation also needs to be provided;
- *sys_call.c*: Defines functions prototypes for system call mapping. Implementation needs to be provided;

The code for Decoder, Encoder and Utils are generic enough, so they are provided as a library. There is no need to have its source code generated.

### D. Translator

The translator input is the binary code of the application we want to execute in the host architecture, in our case PowerPC code. The binary code is loaded from an ELF file [3] of the program to be translated. The Decoder decodes one instruction at a time until a branch instruction is found. Branch instructions are defined in the source architecture description by the command `set_type("jump")`. Figure 9 shows an example of a PowerPC branch instruction definition.

```
bc.set_operands("%imm %imm %addr %imm %imm",
                bo, bi, bd, aa, lk);
bc.set_decoder(opcd=16);
bc.set_type("jump");
```

Figure 9. Conditional branch instruction definition.

Source architecture instructions are decoded to an intermediate representation (IR) and from this into a target architecture IR, as indicated in the mapping description model. The target intermediate representation is then encoded as target binary code. Branch instructions are not translated at this stage; this be done later by the Block Linker. While blocks are not linked, source architecture branch instructions are emulated. Emulation does all the jump test conditions, and register update and next instruction address computation. This emulation sub-system is not generated by the Translator Generator, thus its implementation must be provided by the ISAMAP programmer.

All source architecture registers are represented in memory, thus allowing target and source architectures to have different number of registers. Static one to one register mapping is not allowed. For each reference to source registers in instruction mapping, spill code is generated to load its contents to x86 registers and store them back to memory. A simple but effective register allocation is performed later to improve code performance. Those registers that do not have their content changed in the mapping are considered read only, therefore

[3] *ELF (Executable and Linking Format / Extensible Linking Format)* File standard for executables files, object code, shared libraries and core dumps

they need only to be loaded. Registers that are written are write only, and therefore must be written into memory. If a register is read and have its value changed in the mapping, it must be loaded and written into memory. Spill code is not generated by the translator when we have mappings like the one in figure 6, where the target instruction operand is a memory reference (addr type).

The role of an instruction operand (if it is used or defined) is determined in the architecture description model by key words `set_write` and `set_readwrite`. If neither of them is used, the operand is considered read only. An example is shown in figure 10. Lines 0, 2 and 5 of figure 4 are samples of code generated to load and store register content into memory.

```
add_r32_r32.set_operands("%reg %reg", rm, regop);
add_r32_r32.set_encoder(op1b=0x01, mod=0x3);
add_r32_r32.set_readwrite(rm);

mov_r32_r32.set_operands("%reg %reg", rm, regop);
mov_r32_r32.set_encoder(op1b=0x89, mod=0x3);
mov_r32_r32.set_write(rm);
```

Figure 10. Read/write definition.

*1) Intermediate Representation:* The Intermediate Representation (IR) used by the ISAMAP Translator is the same used in ArchC [14], with some changes to represent instructions, formats and fields. Table I shows the intermediate representation and some description of its fields grouped in a data structure.

Structure `isa_op_field` was included to represent fields that are instruction's operands and operands access mode (read, write, read/write). Field `type` was included to add semantic information about instruction type, as the ArchC language does not support semantics in the description models. Insertion of field `format_ptr` was done to improve the translator performance. Instead of performing a search in a linked list for the format name, we have a pointer to the format object. When an instruction object is created, `format_ptr` receives a pointer to the respective format object. So, instead of performing a linear search ($O(n)$) in a linked list, we have a direct pointer to the desired instruction format ($O(1)$).

Table I
INTERMEDIATE REPRESENTATION

| field | description |
|---|---|
| ac_dec_field | |
| name | field name |
| size | field size in bits |
| first_bit | field first bit position |
| id | field identifier |
| val | field value |
| sign | field sign |
| ac_dec_format | |
| name | format name |
| size | format size in bits |
| fields | format fields |
| ac_dec_list | |
| name | field name |
| value | field value |
| isa_op_field | |
| field | field name |
| writable | access mode (read or write) of the operand assign to the field |
| ac_dec_instr | |
| name | instruction name |
| size | instruction size in bytes |
| mnemonic | instruction mnemonic |
| asm_str | instruction assembly |
| format | instruction format name |
| id | instruction identifier |
| cycles | instruction cycles, not used by ISAMAP |
| min_latency | not used by ISAMAP |
| max_latency | not used by ISAMAP |
| dec_list | field list that identify the instruction |
| cflow | not used by ISAMAP |
| op_fields | fields that represent instruction's operands |
| type | instruction type |
| format_ptr | pointer to instruction format |

## E. Endianness

In binary translation between architectures with different endianness, like the PowerPC (big endian) and x86 (little endian), it is necessary to do endianness conversion every time data is accessed in memory. In PowerPC, endianness conversion is restricted to the translation of load and store instructions.

Before the program starts, the memory region containing data in big endian format is the data segment, where global and static variables are located. Heap segment has no initial values on program start-up and stack is initialized by the ISAMAP Run-Time System. As data can be copied from heap to stack and from stack to heap, the approach adopted by ISAMAP is to handle all data in memory as big endian. The conversion is always done when load/store instructions are executed. The same approach is used by QEMU [11].

Endianness conversion code is specified in the mapping model description for all load/store instructions. A mapping example can be seen in figure 11. 486 processors and later have `bswap` instruction that swap the bytes of 32 bits words, this makes endianness conversion a lot faster. Endianness conversion on 16 bits words is performed by `xchg` instruction.

```
isa_map_instrs {
  lwz %reg %imm %reg;
} = {
  mov_r32_m32 edx $1 $2;
  //Endianness convertion
  bswap_edx;
  mov_r32_r32 $0 edx;
}
```

Figure 11. *Load* instruction mapping.

## F. Run-Time

ISAMAP Run-Time System (RTS) is responsible for initialize the whole environment needed by the translated program execution, code cache management, block linkage and system calls mapping. RTS implementation is very tied to the host

architecture, so its portability is penalized. As an example, context switch between RTS and translated code needs to save and restore all host registers. The code to do this is written in assembly, thus resulting in no portability, but considerably improving performance.

*1) Initialization:* The execution environment of the translated code is set following the source architecture ABI (Application Binary Interface) specifications, in this case PowerPC Linux. Some registers must have an initial value, like PowerPC register R1, which must store the stack pointer. Initialization process is also responsible for allocating the stack. ISAMAP allocates 512 KB stack, as it is sufficient to execute most of SPEC CPU 2000 benchmarks, except 176.gcc, that needs 8 MB of stack size. Stack initial values are also set following ABI specification [16], but they can change according to the environment requirements.

*2) Translated Blocks Execution:* To execute translated code in a binary translator address space, a sandbox is needed. This sandbox must provide two independent environments, so that one does not interfere with the other. Before every translated code execution, all registers used by the translator must be saved (prologue); and after translated code execution, those same registers must be restored (epilogue). Prologue and Epilogue code are shown in figure 12. X86 register `esp` is not saved or restored because it is not used in translated code, avoiding stack issues due to the use of instructions `call` and `ret` to switch between translated code and the Run-Time system. When basic blocks are executed for the first time, the control is switched between RTS and translated code for each basic block. After the first execution of a basic block it is linked by the Block Linkage System and control switching is not needed until another basic block executes for the first time.

| Prologue | Epilogue |
|---|---|
| `mov 0x80a48000, eax` | `mov eax, 0x80a48000` |
| `mov 0x80a48004, ecx` | `mov ecx, 0x80a48004` |
| `mov 0x80a48008, edx` | `mov edx, 0x80a48008` |
| `mov 0x80a4800c, ebx` | `mov ebx, 0x80a4800c` |
| `mov 0x80a48010, esi` | `mov esi, 0x80a48010` |
| `mov 0x80a48014, edi` | `mov edi, 0x80a48014` |
| `mov 0x80a48018, ebp` | `mov ebp, 0x80a48018` |

Figure 12. Prologue and Epilogue code.

*3) Code Cache:* Code translation is much slow when compared with native code execution, so is not a good approach to translate the same basic block for each loop iteration. After each basic block translation and execution, ISAMAP RTS stores it into a code cache. Unlike other DBTs where blocks are interpreted and must reach a threshold before being stored into code cache, ISAMAP stores every executed block. Code cache greatly improves performance by avoiding re-translations. Whenever a block is needed, RTS can fast retrieve it from the code cache.

Translated blocks are identified in code cache by its original address in the original code (before translation). The original address is passed to a hash function and a hash key is obtained. This key will index the hash table. When a block is requested, search in the hash table is performed, what is done very fast. Collisions are solved by chaining other blocks at the same position of the hash table. Figure 13 shows the hash table layout used in ISAMAP.

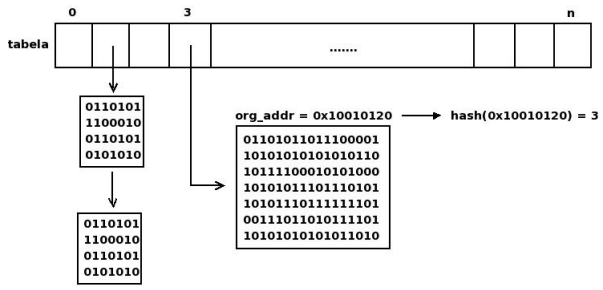


Figure 13. Hash table used in Code Cache

As in QEMU [11], ISAMAP allocates a contiguous memory region of 16MB to be used as code cache. This size is enough to execute all SPEC CPU 2000 benchmarks, but of course, it can be extended. Macro `ALLOC` is used to retrieve the address of the next free position in the code cache to store the translated block. Due to the code cache structure, blocks running in sequence will be next to each other in the code cache, thus improving the translated code performance.

ISAMAP code cache management policy is very simple. Whenever the cache becomes full it is totally flushed, like in QEMU [11]. Code cache is rarely flushed since 16 MB is enough to execute all benchmarks and this approach simplifies the Block Linkage System implementation, as block unlinking becomes unnecessary when the cache is totally flushed.

Some code cache management policies could be used to improve code cache performance. Hazelwood and Smith [17] propose a technique that makes it easy to identify long-lived code blocks to be cached and simultaneously avoid any fragmentation evicting short-lived blocks. Reddi et al. [18] suggested storing and reusing translations across executions, thereby achieving inter-execution persistence.

*4) Block Linker:* To improve performance, the ISAMAP translation system has a block linkage module like QEMU [11] and Yirr-MA [9]. Linking translated blocks avoid control switch between RTS and translated code, improving overall performance.

Block Linker is capable of dealing with four link types: conditional branches, unconditional branches, system calls and indirect branches. System calls are considered unconditional branches in ISAMAP. Assume, for example, that Block A is the last translated and executed block. Block A has two possible successors, Block B and Block C. Block B is the branch instruction's target and Block C is the fall through block. If the next block to be executed is B, Blocks A and B will be linked, otherwise Blocks A and block C will be linked. The remaining block will be linked if and only if it is executed in the future. ISAMAP does block linkage on demand, avoiding that blocks which will never be executed

to be linked and stored into the code cache.

Block linkage is done by adding code stubs at the translated block end. These stubs are needed to test conditions if the branch is conditional, to jump to the successor block, and to return to RTS if the target was not translated yet.

### G. System Calls Mapping

System calls implementation differs for each architecture and/or operational system. The main differences are the number of available calls and parameters, parameter endianness, data structure formats and how parameters are passed. In PowerPC architecture, all system calls parameters are passed in registers, unlike in x86, in which some system calls have parameters passed by memory reference (as few registers are available).

When necessary, ISAMAP System Calls Mapping needs to convert in/out parameters, as well to perform other data conversions to assure the correct execution of the host system calls. In some cases, the values of call's parameters are kernel constants which need to be updated. Some kernel constants have different values for each architecture kernel implementation. An example is the system call `sys_ioctl`, where constants that identify IO devices differ from PowerPC to the x86 kernel implementation. Data structures are another issue, for example, system calls `sys_fstat` and `sys_fstat64` use structs `fstat` and `fstat64`, which have different field alignment in PowerPC and x86 implementations.

In the PowerPC to x86 system call mapping, the six system call parameters (registers R3-R8 in PowerPC) are copied to x86 registers EBX, ECX, EDX, ESI, EDI, EBP. R0 contains system call number, so it is copied to EAX. After the system call execution, EAX content (System call return value) is copied into R0. Before calling a system call, all host registers (except EAX) are saved; after its execution they are restored.

### H. Mapping Improvements

Along the design of ISAMAP, we have noticed that the way instructions are mapped can drastically change the performance of the translated code. As hot code regions are very relevant to program overall performance, mapping should be done very carefully. An example faced during this project was PowerPC instructions that change the register CR (Conditional Register). CR can be changed by arithmetic and logical instructions, and is always changed by comparison instructions. A proper mapping of such instructions has show to be crucial to improve program performance. Figure 14 shows the mapping of `cmp` instruction, which modifies CR; and figure 15 shows a better mapping for the same instruction.

The Conditional Register (CR) is divided in 8 groups of 4 bits. Each group has the following layout from left to right: first bit indicates "less than", second bit indicates "greater than", third bit indicates "equal", and the last bit indicates "summary overflow". The mapping in figure 14 has four branch instructions to change each bit in CR. However, the first three bits are mutual exclusive, in other words, there is no possibility of a logical expression result in "less than", "greater than" or "equal" at same time. Therefore, the mapping of lines 7 to 18 of Figure 14 can be done with lines 5 to 11 of Figure 15 instead. Mapping at figure 15 also has less branch

```
 0  isa_map_instrs {
 1    cmp %imm %reg %reg;
 2  } = {
   ...
 3    mov_r32_r32 ecx src_reg(xer);
 4
 5    mov_r32_imm32 eax #0;
 6
 7    jnz_rel8 #6; // Setting CR[EQ] with ZF
 8    lea_r32_disp32 eax eax #2;
 9
10    jng_rel8 #6;   // Setting CR[GT] to 1
                        if ZF = 0 && SF = OF
11    lea_r32_disp32 eax eax #4;
12
13    jnl_rel8 #6;   // Setting CR[LT] to 1
                        if SF <> OF
14    lea_r32_disp32 eax eax #8;
15
16    and_r32_imm32 ecx #0x80000000;
17    jz_rel8 #6;
18    lea_r32_disp32 eax eax #1;
19
20    mov_r32_imm32 ecx #7;
21    sub_r32_imm32 ecx $0;
22    shl_r32_imm8 ecx #2;
23
24    shl_r32_cl eax;
25
26    mov_r32_imm32 esi #0x0000000f;
27    shl_r32_cl esi;
28    not_r32 esi;
29
30    mov_r32_r32 edi eax;
31
32    and_r32_r32 src_reg(cr) esi;
     // Reseting CR[crfD]
33    or_r32_r32 src_reg(cr) edi;
34 };
```

Figure 14. Cmp instruction mapping.

```
 0  isa_map_instrs {
 1    cmp %imm %reg %reg;
 2  } = {
   ...
 3    mov_r32_m32disp ecx src_reg(xer);
 4
 5    jnl_rel8 #8; // L0
 6    mov_r32_imm32 eax cmpmask32($0, #0x80000000);
 7    jmp_rel8 #13; // L1
//L0:
 8    setg_r8 eax;
 9    movzx_r32_r8 eax eax;
10    lea_r32_sib_disp8 eax eax eax #0 #2;
11    shl_r32_imm8 eax shiftcr($0);
//L1:
12    test_r32_imm32 ecx #0x80000000;
13    jz_rel8 #6;
14    or_r32_imm32 eax cmpmask32($0, #0x10000000);
15
16    and_r32_imm32 src_reg(cr) nniblemask32($0);
17    or_r32_r32 src_reg(cr) eax;
18 };
```

Figure 15. Improved cmp instruction mapping.

instructions, improving performance even more.

As stated above, there are 8 groups of 4 bits to be updated by the `cmp` PowerPC instruction, given it has a parameter that shows which of the 8 groups must be updated with the comparison result. This parameter is an immediate, and thus it does not change throughout the execution. Therefore, the bit mask generated in lines 26-28 of figure 14 can be generated at translation time, which is done only once. To turn this possible, some macros were added to the ISAMAP description language. The macro `nniblemask32()` receives the above mentioned parameter, generates the desired bit mask and puts it into the host instruction. The `and` instruction in line 16 of figure 15 shows an example. This approach eliminates three extras instructions that would be required to build the bit mask. The PowerPC ISA has other instructions which present similar issues related to the bit mask generation from immediate operands; other macros are used to address them.

### *I. Conditional Mapping*

In the PowerPC architecture, the pseudo-instruction `mr` (copy between registers) is implemented by instruction (`or rx ry ry`), which uses the same register for two of its source operands. x86 instruction `or` can be used to directly map `mr` into x86 code. However, the x86 ISA has a specific instruction to copy from one register to another: the `mov` instruction, which is very fast. Thus, using the `mov` instruction to map PowerPC `mr` is better than using the `or` instruction. The mapping will depend on the operands used by the `or` instruction. To address this, a `if-then-else` structure was added to ISAMAP, thus allowing two different mapping to `mr`. The decision is taken on the fly, depending on the `if-then-else` parameters.

Figure 16 shows the PowerPC `or` instruction mapping example, it illustrates the two scenarios listed above. When the two source operands (defined as `rs` and `rb` on the PowerPC models) are the same, the translator emits a mapping related to the true clause of the textttif statement, otherwise, the mapping related to the `else` statement is generated. By adopting this approach, whenever a PowerPC `or` instruction is used to make a copy between registers, it is mapped with one less instruction.

```
isa_map_instrs {
  or %reg %reg %reg;
} = {
  if(rs = rb) {
    mov_r32_m32disp edi $1;
    mov_m32disp_r32 $0 edi;
  }
  else {
    mov_r32_m32disp edi $1;
    or_r32_m32disp edi $2;
    mov_m32disp_r32 $0 edi;
  }
};
```

Figure 16. `Or` instruction mapping.

Another example can be seen in Figure 17. It shows the `Rlwinm` instruction, which rotates source operand to the left, then performs a logical AND with a mask. If the rotate parameter (defined as `sh` on the PowerPC model) is zero, then the x86 instruction `rol` is not needed, resulting in one less instruction in the mapping.

```
isa_map_instrs {
  rlwinm %reg %reg %imm %imm %imm;
} = {
  if(sh = 0) {
    mov_r32_m32disp edi $1;
    and_r32_imm32 edi mask32($3, $4);
    mov_m32disp_r32 $0 edi;
  }
  else {
    mov_r32_m32disp edi $1;
    rol_r32_imm8 edi $2;
    and_r32_imm32 edi mask32($3, $4);
    mov_m32disp_r32 $0 edi;
  }
};
```

Figure 17. `Rlwinm` instruction mapping.

### *J. Run-time Optimizations*

ISAMAP does a few optimization at the basic block level (ISAMAP does not have a trace building mechanism). Optimizations performed are copy propagation, dead code elimination (only `mov` instructions), and local register allocation. Every translated block is optimized. Therefore, almost the whole program code passes through the optimizations. After a block is optimized, it is stored in code cache and set as optimized, leaving the linkage process unchanged.

At first, all source architecture registers are mapped into memory, but with the local register allocation it is possible to exchange memory accesses by registers accesses. Registers are not reallocated, only references to source architecture registers may be allocated to host registers. Memory references to heap, code and stack segments are not considered in the allocation process.

As ISAMAP translates instruction by instruction, it generates unnecessary `load` instructions. For example, in Figure 18 the translation process generated two unnecessary `mov` instructions in lines 3 and 4. Those instructions are removed by the copy propagation optimization. Dead code elimination is used to remove any unnecessary code left by copy propagation.

```
Source PowerPC code

       1.  ADD R1 R2 R3
       2.  SUB R4 R1 R5

Resulting x86 code

       1.   MOV Rtemp R2
       2.   ADD Rtemp R3
       3.   MOV R1 Rtemp
       4.   MOV Rtemp R1
       5.   SUB Rtemp R5
       6.   MOV R4 Rtemp
```

Figure 18. Translation with unnecessary load instructions in lines 3 and 4, which are removed by the optimization process.

## IV. Experimental Results

At the current state, ISAMAP translates and correctly executes the following SPEC CPU 2000 benchmarks: 164.gzip, 175.vpr, 176.gcc, 181.mcf, 186.crafty, 197.parser, 252.eon, 254.gap, 256.bzip2, 300.twolf, 168.wupwise, 171.swim, 172.mgrid, 173.applu, 177.mesa, 178.galgel, 179.art, 183.equake, 187.facerec, 188.ammp, 191.fma3d, 301.apsi. Optimizations are applied only to SPEC INT programs. The following benchmarks are running properly with optimizations: 164.gzip, 175.vpr, 181.mcf, 186.crafty, 197.parser, 252.eon, 254.gap, 256.bzip2, 300.twolf.

### A. Evaluation

Experimental results are presented as follows: ISAMAP performance compared against QEMU performance, ISAMAP optimized performance compared against QEMU performance. The SPEC CPU 2000 reference data set was used; the machine used was a Pentium 4 HT 2.4 GHz, 1 GB RAM. PowerPC code was generated with cross compilers gcc-3.4.5 and gfortran-4.1.0. QEMU version used was 0.11.0.

Figure 19 shows benchmark times for the most effective PowerPC to x86 mapping model, as well as the optimized code times. CP+DC indicates that copy propagation and dead code elimination have been applied, RA indicates that only local register allocation was performed. CP+DC+RA indicates that all optimizations were applied. With all optimizations on, we achieved a maximum speedup of 1.71x on 164.gzip run 2 and only two runs were not benefited by the optimizations (186.crafty run 1 and 252.eon run 1).

Figure 20 shows how ISAMAP compares to QEMU on SPEC INT benchmark. All programs had at least 1.11x speedup; the maximum speedup was 3.16x on 252.eon run 1 with no optimizations. With all optimizations being applied, the maximum speedup achieved was 3.01x on the 252.eon run 3.

Figure 21 shows ISAMAP results, when compared to the QEMU results for the SPEC FLOAT benchmark. Although it is not a fair comparison, we decided to show results for future references. It is not fair to compare these results because ISAMAP uses SSE instructions to translate floating point instructions and QEMU does not. The minimum speedup was 1.79x on 197.art run 1; the maximum speedup was 4.32x on 172.mgrid.

## V. Conclusion

Multi-core processors and 64 bits architectures are quite popular nowadays, and binary translation and emulation have been improving their efficiency to allow improved legacy code execution on these architectures. Binary translation systems also allow optimizations that cannot be done at compile time, due to the lack of execution data behavior.

This paper has presented ISAMAP, a instruction mapping driven by dynamic binary translation. ISAMAP offers a flexible easy-to-use environment to construct instruction mappings between. It has shown better performance when compared to QEMU, reaching up to 3.01x speedups.

Mapping at instruction level has shown to considerably improve overall program performance, as hot code regions can considerably benefit from them. The simplicity of the ArchC constructs of has made it quite easy to design efficient mappings. However, this approach has poor portability when compared with QEMU [11], and Yirr-MA [9], where mappings are described at a higher abstraction level.

We believe that our work provides a convenient and detailed development environment for further DBT programmers. In order to extend the system to target other architectures, there is no need of knowledge of the detailed implementation of the translator, only source/target ISA descriptions and a mapping between then are needed.

### A. Future Works

Dynamic optimizations have the potential to improve legacy code performance, given the availability of run time information. PIN [19], is an efficient tool for code analysis and instrumentation, which has been extensively used to study program dynamic behavior [18], [20] and [21]. Several papers present efficient binary translation systems which apply dynamic optimizations to improve sequential code execution (e.g. [8], [22] and [23]) and parallelization (e.g. [24], [25] and [26]).

For the future, we intend to implement more sophisticated optimizations techniques, based on trace construction, and dynamic code parallelization, mainly due to multi-core architecture popularization, which demands multi-thread application to use the whole hardware potential. We also intend to expand ISAMAP, making it capable of dealing with self-modifying code and exceptions. We believe ISAMAP can also be used as a way to automatically synthesize code mapping fragments, so as to explore the potential of low-level ISA descriptions.

## References


[1] V. J. R. D. G. Tipp Moseley, Alex Shye and R. Peri, "Shadow Profiling: Hiding Instrumentation Costs with Parallelism." Proceedings of the International Symposium on Code Generation and Optimization CGO 07, March 2007.

[2] S. Wallace and K. Hazelwood, "SuperPin: Parallelizing Dynamic Instrumentation for Real-Time Performance." Proceedings of the International Symposium on Code Generation and Optimization CGO 07, March 2007.

[3] C. Zheng and C. Thompson, "PA-RISC to IA-64: Transparent Execution, No Recompilation," *IEEE Computer*, vol. 33, no. 3, pp. 47–53, March 2000.

[4] A. Chernoff and R. Hookway, "DIGITAL FX!32 - Running 32-Bit x86 Applications on Alpha NT," USENIX Association. Berkeley CA: Proceedings of the USENIX Windows NT Workshop, August 1997.

[5] L. Baraz, T. Devor, O. Etzion, S. Goldenberg, A. Skaletsky, Y. Wang, and Y. Zemach, "IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems." San Diego CA: 6th International Conference on Microarchitecture (MICRO36), December 2003.

[6] V. Bala, E. Duesterwald, and S. Banerjia, "Dynamo: A transparent Dynamic Optimization System," *SIGPLAN PLDI*, pp. 1–12, June 2000.

[7] D. Ung and C. Cifuentes, "Optimising hot paths in a dynamic binary translator." In Workshop on Binary Translation, October 2000.

[8] J. Lu, H. Chen, P.-C. Yew, , and W.-C. Hsu, "Design and implementation of a lightweight dynamic optimization system," *The Journal of Instruction-Level Parallelism*, vol. 6, 2004.

[9] P. J. and G. J., "Fast dynamic binary translation the yirr-ma framework." In Proceedings of the 2002 Workshop on Binary Translation, 2002.

| Benchmark | Run | isamap time (s) | cp+dc | | ra | | cp+dc+ra | |
|---|---|---|---|---|---|---|---|---|
| | | | time (s) | speedup | time (s) | speedup | time (s) | speedup |
| 164.gzip | 1 | 270.63 | 174.65 | 1.55 | 166.59 | 1.62 | 162.26 | 1.67 |
| | 2 | 119.88 | 83.47 | 1.44 | 73.32 | 1.64 | 69.84 | 1.72 |
| | 3 | 255.22 | 214.27 | 1.19 | 187.44 | 1.36 | 185.27 | 1.38 |
| | 4 | 199.80 | 167.54 | 1.19 | 143.07 | 1.40 | 140.45 | 1.42 |
| | 5 | 524.48 | 337.74 | 1.55 | 331.99 | 1.58 | 320.75 | 1.64 |
| 175.vpr | 1 | 713.41 | 680.04 | 1.05 | 664.75 | 1.07 | 631.38 | 1.13 |
| | 2 | 473.28 | 449.59 | 1.05 | 436.25 | 1.08 | 412.88 | 1.15 |
| 181.mcf | 1 | 439.89 | 429.24 | 1.02 | 419.05 | 1.05 | 411.06 | 1.07 |
| 186.crafty | 1 | 1144.83 | 1206.99 | 0.95 | 1255.53 | 0.91 | 1200.25 | 0.95 |
| 197.parser | 1 | 1380.80 | 1245.55 | 1.11 | 1075.89 | 1.28 | 1039.24 | 1.33 |
| 252.eon | 1 | 567.73 | 593.48 | 0.96 | 605.24 | 0.94 | 673.01 | 0.84 |
| | 2 | 432.11 | 451.97 | 0.96 | 397.52 | 1.09 | 416.94 | 1.04 |
| | 3 | 789.38 | 791.23 | 1.00 | 792.04 | 1.00 | 779.71 | 1.01 |
| 254.gap | 1 | 1066.51 | 994.65 | 1.07 | 805.54 | 1.32 | 799.19 | 1.33 |
| 256.bzip2 | 1 | 351.81 | 324.16 | 1.09 | 277.55 | 1.27 | 259.19 | 1.36 |
| | 2 | 413.28 | 385.47 | 1.07 | 331.08 | 1.25 | 309.45 | 1.34 |
| | 3 | 363.45 | 337.17 | 1.08 | 289.36 | 1.26 | 273.71 | 1.33 |
| 300.twolf | 1 | 1662.39 | 1634.97 | 1.02 | 1456.39 | 1.14 | 1441.34 | 1.15 |

Figure 19. ISAMAP X ISAMAP OPT SPEC INT

| Benchmark | Run | qemu time (s) | isamap | | cp+dc | | ra | | cp+dc+ra | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | | time (s) | speedup | time (s) | speedup | time (s) | speedup | time (s) | speedup |
| 164.gzip | 1 | 260.09 | 270.63 | 0.96 | 174.65 | 1.49 | 166.59 | 1.56 | 162.26 | 1.60 |
| | 2 | 151.70 | 119.88 | 1.27 | 83.47 | 1.82 | 73.32 | 2.07 | 69.84 | 2.17 |
| | 3 | 319.75 | 255.22 | 1.25 | 214.27 | 1.49 | 187.44 | 1.71 | 185.27 | 1.73 |
| | 4 | 298.25 | 199.80 | 1.49 | 167.54 | 1.78 | 143.07 | 2.08 | 140.45 | 2.12 |
| | 5 | 531.72 | 524.48 | 1.01 | 337.74 | 1.57 | 331.99 | 1.60 | 320.75 | 1.66 |
| 181.mcf | 1 | 506.01 | 439.89 | 1.15 | 429.24 | 1.18 | 419.05 | 1.21 | 411.06 | 1.23 |
| 186.crafty | 1 | 1338.54 | 1144.83 | 1.17 | 1206.99 | 1.11 | 1255.53 | 1.07 | 1200.25 | 1.12 |
| 197.parser | 1 | 1716.82 | 1380.80 | 1.24 | 1245.55 | 1.38 | 1075.89 | 1.60 | 1039.24 | 1.65 |
| 252.eon | 1 | 1796.67 | 567.73 | 3.16 | 593.48 | 3.03 | 605.24 | 2.97 | 673.01 | 2.67 |
| | 2 | 1240.23 | 432.11 | 2.87 | 451.97 | 2.74 | 397.52 | 3.12 | 416.94 | 2.97 |
| | 3 | 2349.40 | 789.38 | 2.98 | 791.23 | 2.97 | 792.04 | 2.97 | 779.71 | 3.01 |
| 254.gap | 1 | 1142.63 | 1066.51 | 1.07 | 994.65 | 1.15 | 805.54 | 1.42 | 799.19 | 1.43 |
| 256.bzip2 | 1 | 415.36 | 351.81 | 1.18 | 324.16 | 1.28 | 277.55 | 1.50 | 259.19 | 1.60 |
| | 2 | 466.29 | 413.28 | 1.13 | 385.47 | 1.21 | 331.08 | 1.41 | 309.45 | 1.51 |
| | 3 | 416.24 | 363.45 | 1.15 | 337.17 | 1.23 | 289.36 | 1.44 | 273.71 | 1.52 |
| 300.twolf | 1 | 2051.37 | 1662.39 | 1.23 | 1634.97 | 1.25 | 1456.39 | 1.41 | 1441.34 | 1.42 |

Figure 20. ISAMAP X QEMU SPEC INT

| Benchmark | Run | qemu time (s) | isamap time(s) | speedup |
|---|---|---|---|---|
| 168.wupwise | 1 | 1555.180 | 540.740 | 2.88x |
| 172.mgrid | 1 | 3533.060 | 818.010 | 4.32x |
| 173.applu | 1 | 2189.560 | 531.850 | 4.12x |
| 177.mesa | 1 | 1252.550 | 691.570 | 1.81x |
| 178.galgel | 1 | 1678.140 | 671.290 | 2.50x |
| 197.art | 1 | 163.670 | 91.310 | 1.79x |
| | 2 | 180.010 | 100.140 | 1.80x |
| 183.equake | 1 | 682.760 | 257.470 | 2.65x |
| 187.facerec | 1 | 1562.720 | 427.160 | 3.66x |
| 188.ammp | 1 | 2708.610 | 768.380 | 3.53x |
| 191.fma3d | 1 | 2241.020 | 949.710 | 2.36x |
| 301.apsi | 1 | 2004.340 | 707.170 | 2.83x |

Figure 21. ISAMAP X QEMU SPEC FLOAT

[10] K. Ebcioğlu and E. R. Altman, "Daisy: dynamic compilation for 100architectural compatibility," in *ISCA '97: Proceedings of the 24th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 1997, pp. 26–37.

[11] F. Bellard, "Qemu, a fast and portable dynamic translator," in *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2005, pp. 41–41.

[12] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang, "Code generation using tree matching and dynamic programming," *ACM Trans. Program. Lang. Syst.*, vol. 11, no. 4, pp. 491–516, 1989.

[13] C. Fraser, D. Hanson, and T. Proebsting, "Engineering a Simple, Efficient Code Generator Generator," *ACM Letters on Prog. Lang. and Systems*, pp. 213–226, 1993.

[14] R. Azevedo, S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros, "The ArchC architecture description language and tools," *International Journal of Parallel Programming*, vol. 33, no. 5, pp. 453–484, October 2005.

[15] A. Baldassin, P. C. Centoducatte, and S. Rigo, "Extending the ArchC language for automatic generation of assemblers," in *Proceedings of the 17th Symposium on Computer Architecture and High Performance Computing*, October 2005, pp. 60–68.

[16] "Linux Standard Base Specification for the PPC32 Architecture 1.3," Free Standards Group, Tech. Rep., http://refspecs.freestandards.org/LSB_1.3.0/PPC32/spec.html last accessed February 18, 2010.

[17] K. Hazelwood and M. D. Smith, "Managing bounded code caches in dynamic binary optimization systems," *ACM Trans. Archit. Code Optim.*, vol. 3, no. 3, pp. 263–294, 2006.

[18] V. J. Reddi, D. Connors, R. Cohn, and M. D. Smith, "Persistent code caching: Exploiting code reuse across executions and applications," in *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 74–88.

[19] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*. New York, NY, USA: ACM, 2005, pp. 190–200.

[20] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *CF '07: Proceedings of the 4th international conference on Computing frontiers*. New York, NY, USA: ACM, 2007, pp. 143–152.

[21] K. Hazelwood and R. Cohn, "A cross-architectural interface for code cache manipulation," in *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 17–27.

[22] J. Dehnert, B. Grant, J. Banning, R. Johnson, T. Kistler, A. Klaiber, , and J. Mattson, "The Transmeta code morphing software: Using speculation, recovery, and adaptive retranslation to address reallife challenges." In International Symposium on Code Generation and Optimization. IEEE Computer Society, 2003, pp. 15–24.

[23] S. X. B. H. Jianhui Li, Qi Zhang, "Optimizing Dynamic Binary Translation for SIMD Instructions." Proceedings of the International Symposium on Code Generation and Optimization CGO 06, March 2006.

[24] E. Yardimci and M. Franz, "Dynamic Parallelization and Mapping of Binary Executables on Hierarchical Platforms." Proceedings of the 3rd conference on Computing frontiers CF 06, May 2006.

[25] C. K. Jisheng Zhao and I. Rogers, "Lazy Interprocedural Analysis for Dynamic Loop Parallelization." Workshop on New Horizons in Compilers, Bangalore, India, December 2006.

[26] C. Wang, Y. Wu, E. Borin, S. Hu, W. Liu, D. Sager, T.-f. Ngai, and J. Fang, "Dynamic parallelization of single-threaded binary programs using speculative slicing," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*. New York, NY, USA: ACM, 2009, pp. 158–168.