SHRINK: Reducing the ISA Complexity Via Instruction Recycling

Bruno Cardoso Lopes Rafael Auler Luiz Ramos Edson Borin Rodolfo Azevedo
University of Campinas - UNICAMP - Brazil
{blopes,auler,luiz.ramos,edson,rodolfo}@ic.unicamp.br

Abstract

Microprocessor manufacturers typically keep old instruction sets in modern processors to ensure backward compatibility with legacy software. The introduction of newer extensions to the ISA increases the design complexity of microprocessor front-ends, exacerbates the consumption of precious on-chip resources (e.g., silicon area and energy), and demands more efforts for hardware verification and debugging. We analyzed several x86 applications and operating systems deployed between 1995 and 2012 and observed that many instructions stop being used over time, and more than 500 instructions were never used in these applications. We also investigate the impact of including these unused instructions in the design of the x86 decoders and propose SHRINK, a mechanism to remove old instructions without breaking backward compatibility with legacy code. SHRINK allows us to remove 40% of the instructions from the x86 ISA and improve the critical path, area, and power consumption of the instruction decoder, respectively, by 23%, 48%, and 49%, on average.

1. Introduction

Adding new instructions to an ISA is a complex task with undesired side effects. Processor manufacturers typically seek to maintain backward compatibility with legacy software stacks, and the binary encodings used to represent instructions assume a fixed number of bits. Therefore, maintaining legacy instructions while adding new ones eventually (1) exhausts the range of unique instruction representations (opcode space). Even when the opcode space has not yet been exhausted, new instructions tend to be encoded with longer bit strings than legacy instructions. Thus, when compilers adopt newer instructions, (2) binaries get larger. Furthermore, processors replicate instruction decoders to increase throughput, which demands a lean decoder design to save area in spite of the fact that new ISA additions (3) increase decoder complexity. A more complex instruction decoder, in turn, affects (4) performance [31] and (5) power. In summary, the instruction cache (I-cache) suffers increasing pressure and instruction decoders grow, reaching up to 18% [25, 26] of the chip area.

Given those observations, there is a clear trade-off between the ability to extend microprocessor ISAs efficiently and the ability to maintain backward compatibility. We refer to this trade-off as **the ISA aging problem**, which processor manufacturers address using different approaches. Modern x86 processors add instruction prefixes and incorporate alternative processor modes. The prefixes expand the opcode space to accommodate new instructions, but increase the length of new instructions relative to that of old instructions. The alternative processor modes enable the reinterpretation of 32-bit instructions in the context of 64-bit programs [16, 28], but add complexity to the processor design. Conversely, the IBM POWER ISA admits some loss of backward compatibility in trade for cost and performance optimizations. Specifically, the manufacturer may release market-driven processor implementations that add instruction categories or delete deprecated instructions whenever the benefit of the optimizations outweighs the loss in compatibility [15].

We propose SHRINK, an approach to address the harmful effects of ISA aging and reduce the ISA complexity without compromising backward code compatibility. SHRINK provides a method for recycling instructions (i.e., removing unused and/or infrequently-used instructions and strategically reassigning their encodings to more frequently used instructions), and a mechanism to emulate the removed instructions. SHRINK enables the conception of ISA revisions that are as efficient, in code compaction and decoder sizes, as newly designed ISAs, while retaining backward compatibility.

The contributions of this work are as follows. We performed an extensive chronological analysis of several x86 applications and operating systems and show that several instructions are rarely used or become unused over time. We evaluate how the unused instructions affect the size, power, and performance of x86 instruction decoders. We propose SHRINK and evaluate how it improves the x86 instruction decoder and the I-cache miss rates of SPEC CPU 2006 benchmarks.

This paper is organized as follows: Section 2 discusses ISA aging; Section 3 studies radical approaches to address the related aging issues; Sections 4, 5, and 6 describe SHRINK, its implementation, and its evaluation, respectively. Finally, Sections 7 and 8 present related work and our conclusions.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
ISCA '15, June 13 - 17, 2015, Portland, OR, USA
© 2015 ACM. ISBN 978-1-4503-3402-0/15/06...$15.00
DOI: http://dx.doi.org/10.1145/2749469.2750391

2. ISA Aging

In this section, we initially discuss a key aspect of the ISA aging problem: the growth of instructions, in number and in length. Then, we inspect x86 instructions usage over time.

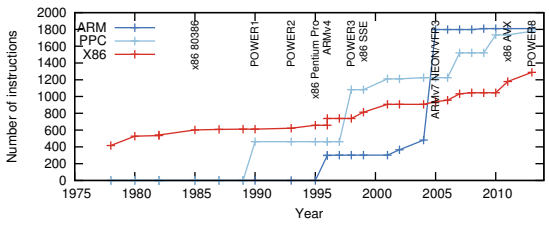


Figure 1: ISA growth (in number of instructions) over time for different processor architectures

Growth in the number of instructions. In Figure 1, we observe a steady growth in the number of instructions in the x86, POWER, and ARM ISAs. In the x86 ISA, the 8086 processor (from 1978) had about 400 instructions. That number had doubled by 1999, with the release of Pentium III and the SSE multimedia extension [23]. Today, the ISA has reached 1300 instructions in the Haswell architecture [18] and in the near future, with Intel's Knights Landing architecture supporting AVX512, is expected to include more 100 new instructions [8]. There is no evidence that the ISA will stop growing. Similarly, the number of instructions in the first POWER ISA doubled in the POWER3 ISA (from 1998), which incorporates instructions from the PowerPC, POWER1, and POWER2 ISAs. More recently, the initial number reached a threefold growth, in the POWER6 ISA and its AltiVec SIMD extension (from 2007), and continues to increase in the POWER8 ISA (from 2013). Interestingly, the ISA retires instructions across generations to enable cost and performance optimizations [15]. Finally, Figure 1 also shows that the ARM ISA grew – from 300 in V4 (from 1996) to nearly 1800 instructions, with the addition of the NEON Advanced SIMD extensions (from 2005). Moreover, ARM's Thumb extension also supports NEON/VFP, increasing the likelihood that the ISA will suffer with aging.

Unique instruction signatures (UIS). In the x86 ISA, the length and the format of instructions may vary. To minimally identify an instruction, we must match its *opcode* field. However, most instructions require an extra *prefix* field, and others require a *ModR/M* field. For that reason, we adopt the term *unique instruction signature (UIS)*, which denotes the minimum combination of those fields that are necessary to identify an instruction. For example, instruction `addl Id, Rd` (add a 32-bit immediate to a 32-bit register) has two different opcodes, depending on whether `Rd` is `%eax` or not. In this case, we would have two different UISes. With that in mind, we define *UIS space* as the set of all valid ISA binary encodings.

Growth in instruction length. A size-efficient approach for a CISC ISA organization would assign the most frequent

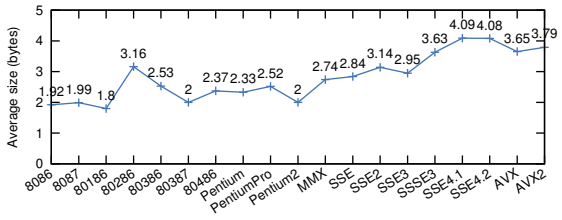


Figure 2: Average UIS size over number of instructions for x86 for each ISA extension

instructions to the smallest UISes. However, currently, new instructions tend to accumulate extra fields to differentiate them from the old ones. For example, in our study, we detected that the IA-32 instruction `AAA` (ASCII adjust after addition) is never found in modern software but occupies a prime area in the opcode space (1-byte opcode). In contrast, the instruction `vcmpsd` (from the AVX IA-32 extension [17]) uses 5 bytes for opcode and is often used by modern compilers to generate floating-point computations. As a result of this trend, the average instruction size increases. Moreover, as new instructions grow in popularity, the resulting binary executables decrease in space efficiency. Figure 2 shows the increase in the average size of UIS in each x86 ISA extension over the years. For example, 308 new instructions were introduced between MMX and SSE 4.2 as shown by Figure 1, and to represent this additional amount of instructions, more than 1 UIS byte is needed. Therefore the average number of UIS bytes has grown from 2.7 to 4 bytes (Figure 2).

The impact of ISA aging. In modern x86 implementations, such as the Haswell microarchitecture, an instruction traverses several components before being executed, namely: (1) an instruction fetch unit, (2) an I-cache, (3) a 16B predecode fetch buffer, (4) an instruction queue, and (5) one within a set of decoders [21]. Due to ISA aging, as instruction length grows, we observe three main effects. First, fewer instructions fit into the I-cache, thereby reducing its effective capacity [31]. Second, the predecode buffer either needs to increase in size (therefore, in latency), be replicated, or otherwise reduce the instruction-decoding throughput. Third, the variety of instruction length imposes undesirable sequentiality to the decoding process, since leading instructions need to be decoded and have their length determined before the next instruction can be detected [27]. As a result, instruction decoders tend to become a bottleneck, as pipelines widen. In summary, having fewer, shorter instructions, would simplify the processor's design, testing and validation, the effectiveness of the I-cache, and the final size of the binary executables.

2.1. Instruction Usage Over Time

We presented several aspects of the x86 ISA growth history. We show now how often each one of the x86 instructions is used in practice. Concretely, we capture how programs spanning multiple generations and systems utilize the x86 ISA for common user tasks (e.g., software used in home or office

Release	Operating System	Additional Software
1996-1997	Slackware Linux 3	Netscape 4.0.1, StarOffice 3.1
2003-2004	Ubuntu 4.10	Firefox 0.9.2, OpenOffice 1.1.2
2005-2006	Ubuntu 6.10 *	
2006-2007	Ubuntu 7.10 *	
2007-2008	Ubuntu 8.10	Firefox 3.0.3, OpenOffice 2.4
2009-2010	Ubuntu 10.10 *	
2011-2012	Ubuntu 12.04	Firefox 11, LibreOffice 3.5
1995-1996	Windows 95	I.E. 3, Office 95
1998-2000	Windows 98 SE	I.E. 5, Office 2000
2001-2004	Windows XP SP2	I.E. 6, Office 2003
2007-2009	Windows Vista	I.E. 7, Office 2007
2010-2012	Windows 7 SP1	I.E. 8, Office 2010

Table 1: Virtual machine setup (* = static analysis only).

Type	1 Byte	2 Bytes	3 Bytes	4 Bytes	5 Bytes	6 Bytes
AVX	0	0	3	61	5	0
SSE	0	0	74	238	7	1
Other	0	0	40	76	0	0
Total	0	0	117	375	12	1

Table 2: Number of unused UISes, grouped by size.

computers). We organized several virtual machines, each one containing a complete 32-bit x86 software stack from a specific year, as summarized in Table 1. For instance, our first Windows-based stack contains a full installation of Windows 95, Internet Explorer 3 (browser), and Office 95 (productivity suite) to represent how x86 software from 1995 to 1996 used the IA-32 ISA.

Because instructions may never be executed (in spite of being catalogued), we initially perform a static analysis of the disk images of our virtual machines. Specifically, we implement an instruction crawler that searches each disk image entirely and counts how many times each UIS appears in any executable file. Our crawler uses two disassembling tools as libraries: Agner's *object file converter* [9] tool and the disassembler library of the `Bochs` virtual machine [24].

Conversely, since instructions that do not appear in our static analysis may be generated and executed at runtime, we also perform a dynamic analysis of our virtual machines. In particular, we use a modified version of `Bochs` virtual machine that reports histograms of executed instructions. With `Bochs`, we recorded a trace of common user activities, such as booting up and shutting down the operating system; starting text editors for editing and formatting large texts; starting spreadsheet for loading, sorting, and processing large files containing floating-point numbers; opening local copies of websites on an Internet browser; and decompressing large files. Because `Bochs` was unable to run Ubuntu 6, 7 and 10, we do not include them in our dynamic analysis.

Total unused instructions. Table 2 breaks down the number of unused UISes by size. We consider instructions that belong to vector extensions in separate categories, because albeit unused, they are still under adoption and their UISes are likely to be used in the future. SSE includes all Intel SSE and AMD SIMD extensions; AVX includes the AVX and AVX2 extensions; and Others include the MMX extension. In total, we found that 505 UISes are unused, i.e., about 30% of all 1646 x86 prefix and UIS combinations never appeared in our static analysis. Note that this result does not account for 149 combinations that either requires the 64-bit mode or execution privileges, since our analysis focuses on 32-bits VMs.

When our instruction crawler sees a UIS in software released at a given year (e.g., 2004), but cannot find it in software released in the subsequent years (2006 to 2012), we consider this UIS as unused or outdated. Next, we discuss UISes that became unused over time according to the static (disk analysis) and dynamic (runtime analysis) approaches.

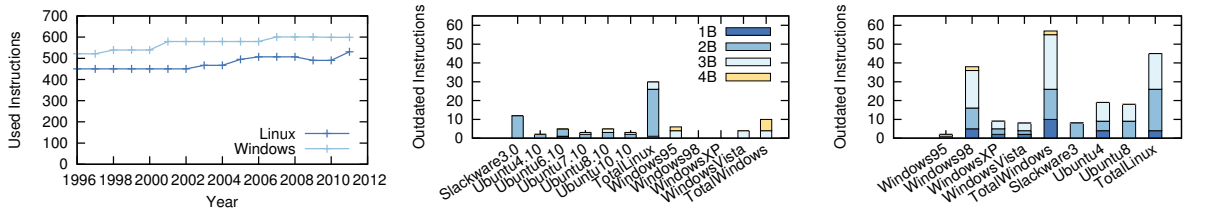
Static analysis. Figure 3b shows the number of UISes that became unused over time, categorized by size. In Slackware, for instance, we see that 12 2-byte UISes were last seen in 1996. This means that subsequent software releases stopped using those instructions. Not surprisingly, some outdated UISes discovered by our static analysis were also deprecated in Intel IA-32e 64-bit mode, including `les`, load far pointer using ES, `push` and `pop` using ES or CS. They were outdated starting with Ubuntu 7.

Figure 3a depicts the number of UISes used in the Linux and Windows systems over time. As expected, that number increases because newer programs eventually adopt the new instructions. This is the case for several `cmov` (conditional move) variants, which were first introduced in Pentium Pro. Likewise, `vmclear` and `vmptrld` became more popular as virtualization became more pervasive, and `xchg` and `xadd` (exchange and add instructions), with the rise of multicore.

Dynamic Analysis Our dynamic analysis is less conservative than our static analysis, but it helps uncovering the instructions used at runtime by programs in different activities and with different inputs. For example, we were able to find instructions that were generated by self-modifying code or as part of a JIT engine. However, those categories of dynamic instructions were small compared to the amount of UISes that are present in disk images but are never used at runtime. As in our static analysis, we did not consider vector, privileged, and 64-bit instructions.

Figure 3c shows how instructions became unused over time in our dynamic analysis. We observe, for instance, that Windows 98 uses 38 UISes that were not found in dynamic traces of subsequent Windows releases. Interestingly, our dynamic analysis found a much larger number of unused instructions than the static analysis, which hints that although instructions stop being used, they may still appear in disk images.

We verify an interesting aspect of the ISA aging problem by looking at how many single-byte instructions became unused over time. Single-byte instructions are strategic because if they are used frequently enough, as compared to the other larger instructions, then the resulting binary executables will be efficient in size. In our analysis of the Windows-based systems, we found that 10 single-byte UISes stopped being used from Windows 95 up to Windows 7. In fact, in Windows, many UISes that were considered to be frequently used in our static analysis proved to be unused in our dynamic analysis.



(a) UISes found on disk for each OS, by year (b) Static analysis of outdated UISes (c) Dynamic analysis of outdated UISes

Figure 3: Static and dynamic instruction usage patterns for Windows and Linux systems. For static analysis, we considered all binaries in the virtual machine file system. For dynamic analysis, we executed a common set of tasks in each virtual machine.

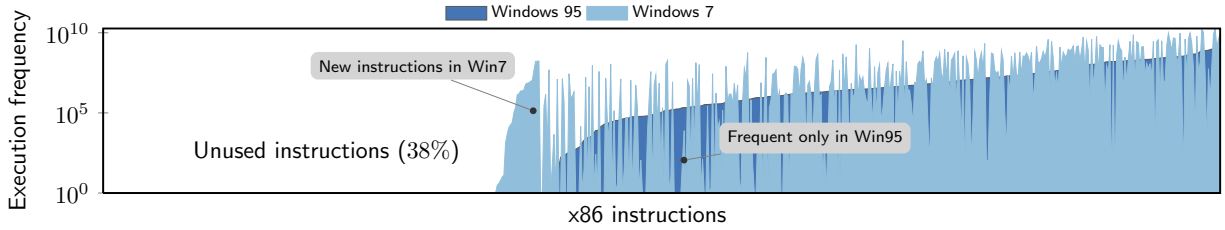


Figure 4: Histogram (in logarithmic scale) of dynamic instructions sorted by frequency with respect in Windows 95 and compared to their corresponding frequency in Windows 7. Spikes show differences in the usage pattern.

This suggests that Windows may keep many old libraries in disk for compatibility. In the Linux-based systems, only 4 single-byte UISes were unused from Slackware 3 to Ubuntu 12. As the chart shows, Ubuntu 6.10 was the last release to use the single-byte UIS instruction `les`.

To show that instructions have well-defined lifetimes, in which programs frequently use them in the past and then cease their usage afterwards, Figure 4 shows a dynamic instruction frequency histogram in logarithm scale for 2 traces: the Windows 95 (1995) workload and the Windows 7 (2010) workload. The x-axis represents different instruction types and the y-axis shows their corresponding usage frequency. The UISes are ordered by frequency count in the oldest workload. The fact that the Windows 7 has frequency peaks and valleys is evidence that many popular instructions in Windows 95 had a very different usage pattern 15 years later. They either stopped being used or had their usage reduced.

3. Radical Approaches

In this section, we present an exploratory evaluation of radical approaches for the ISA aging problem. Specifically, we discuss and evaluate the impact of three radical approaches for creating space in the x86 ISA. All approaches fully re-encode the ISA and possibly remove unused instructions, thereby violating backward compatibility. Note that in practice, maintaining that compatibility would require extensive and complex modifications to the virtualization support, binary translation, and compiler infrastructure, to mention a few challenges.

Approach 1: reduce all UISes to 2 bytes. Instead of having variable-size UISes, we encode all UISes into 2 bytes, while maintaining registers and immediates with the same

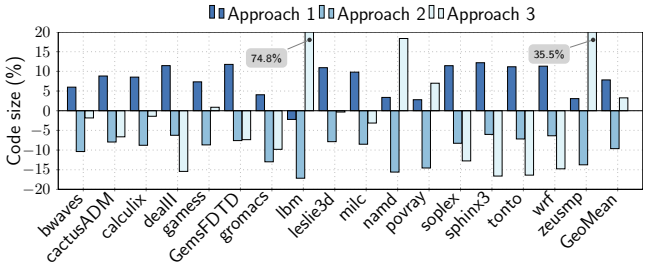


Figure 5: SPEC CFP 2005 re-encoded with three approaches.

encoding as before. As a result, the ISA supports up to 64K instructions with plenty of room for new instructions.

Approach 2: reduce all UISes to 1 or 2 bytes. This approach is similar to the previous one. We reserve 1-byte for the 240 most-used instructions and 2-bytes for the other instructions, while maintaining the registers and immediates with the same encoding as before. To determine the number of 1-byte instructions, we evaluated the resulting code size when encoding from 10 to 240 instructions with one byte. We found that the diminishing return point is around 50 instructions, but we decided to use 240 because it still leaves room for adding a few new 1-byte instructions. The resulting ISA has up to 4336 instructions, and leaves room for new instructions.

Approach 3: convert to a RISC-like ISA encoding. In this case, all instructions have 32-bit encodings. For simplicity, we replace the x86 encoding with the ARM ISA encoding.

Using SPEC CPU 2006[13], we evaluate the three approaches with respect to the size of binary executables (code size) that they generate. In Figure 5, we show results for the floating-point benchmarks only, since the integer benchmarks exhibit similar results. Our main finding is that the x86 code

size is larger than that of its ARM counterpart for most programs (but not in geometric mean). We did not expect this result, since ARM is a RISC ISA (known to have bigger footprints than CISC ISAs). In addition, we observe that despite limiting the UIS space, Approach 2 (variable-length UISes) presents the best reduction in code size. To investigate the impact of the best radical approach on cache misses, we re-encoded 5 SPEC 2006 benchmarks using Approach 2 and simulated the I-cache behavior using a 32KB, 4-way configuration. We found cache miss reductions of: 7% for mcf, 5% for omnetpp, 49% for lbm, 6% for astar, and 19% for milc. Finally, we compare the two CISC-like re-encoding approaches (1 and 2) to the current x86 ISA across two modern operating systems (Windows 7 and Ubuntu 12). We observe that compared to the current x86 ISA, Approach 1 increases code size by 70% on Windows and 65% on Ubuntu, whereas Approach 2 reduces code size by 15% on Windows and 17% on Ubuntu.

4. Instruction Recycling in SHRINK

In this section, we provide an overview of SHRINK's instruction recycling mechanism. The recycling mechanism enables removing old instructions from the ISA and emulating them so as to retain compatibility with software that uses removed instructions. Likewise, the mechanism makes it possible for older processors to run software that uses new instructions.

Revisions. We first define *processor revision (PR)* and *software revision (SR)*. A PR denotes an ISA version implemented in a CPU, whereas SR identifies the PR for which a program was compiled. For example, software at revision SR_3 running on a processor at revision PR_2 may eventually attempt to use instructions not yet implemented in that CPU. Conversely, software at revision SR_2 running on a processor at revision PR_3 may use instructions that were removed from that CPU. Therefore, software at SR_i is fully compatible with a processor at PR_j only if $i = j$.

Instruction Lifetime. We traditionally expect a UIS to be tightly related with its machine operation. For instance, x86 programmers may instinctively associate the opcode `0xCC` with a trap that triggers the interrupt number 3. However, we want to dissociate instructions from UISes. Hence, we define *instruction lifetime* as the limited period of time in which an instruction is associated with and incarnates a particular UIS.

Definition of Recycling. In SHRINK, instruction recycling is the process by which an instruction ends its lifetime and is subsequently replaced by a new instruction. In the process, the old instruction departs from its current UIS, which then becomes associated with the new instruction. This process may repeat itself multiple times, and a UIS may span the lifetime of many different instructions.

4.1. Instruction Lifecycle

As shown in Figure 6, an instruction's lifecycle can be divided in three parts: Creation, Outdating, and Recycling.

Creation. Initially, a new instruction is associated with a UIS at PR_A. That UIS must have either never been used before (e.g., a previously invalid or reserved opcode), or have been recycled from another instruction. Because adopting the new instruction in compiler back-ends and vendor libraries takes time, only after a significant software evolution cycle the new instruction is incorporated into user programs.

Outdating. As processors evolve with technological advances and new business demands, new features may supersede the functionality of obsolete instructions. Upon deciding to deprecate an instruction, the manufacturer announces this relevant fact to the community, which happens some time between PR_A and PR_B. Eventually, compiler manufacturers remove support for that instruction and operating systems vendors start working on emulation routines for the instruction.

To decide whether to remove or keep an instruction in the ISA, we propose a recycling policy parameterized by α, β, and Δ (as shown in Figure 6). α is a grace period for new ISA instructions, i.e., the time given to compiler and software developers to adopt the new instructions. No instruction created in this period is considered for removal. β represents the Outdating period. We consider outdated all instructions that are not used within the α window, and we give this time to developers to remove them from their software base. After β elapses, instructions can be removed from the hardware and should start being emulated in software. Finally, Δ is the minimum instruction lifespan ($\Delta = \alpha + \beta$).

Recycling. An old instruction to be removed departs from its current UIS in PR_B, and a new instruction becomes associated with that UIS. Because a processor holds thousands of UISes and the recycling and creation steps occur simultaneously within a same PR, a SHRINK-enabled CPU must internally be able to associate a UIS with a new instruction without impacting other UIS-instruction associations. To enable that feature, SHRINK provides each UIS with its own *UIS Revision (UR)*, which keeps track of how many instructions have been associated with a specific UIS over time. To ensure consistency, a change in any UR causes a change in PR.

Figure 7 illustrates the UR updating process. In the figure, instruction *AAA* (UIS 0x00) becomes obsolete. After an outdating period, in PR_A, UIS 0x00 is disassociated from instruction *AAA* and recycled for association with the new instruction *VADD*. Before PR_A, the UR of UIS 0x00 is 0. After the recycling occurs, the UR of UIS 0x00 becomes 1, whereas the URs of other UISes remain unaffected.

Backwards compatibility. SHRINK relies on software emulation for instructions that are not implemented in a given PR by generating CPU traps. The SR is annotated in software binaries, and the processor generates a trap for instructions such that $UR[PR] \neq UR[SR]$.

Revision Vectors and The Trap Mask. We define *Revision Vector (RV)* as the set of URs of a processor. Given PR_x and N, the total number of UIS, RV_x is defined as:

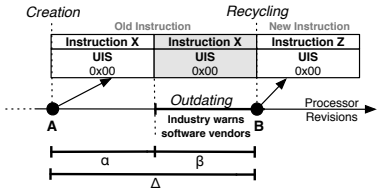


Figure 6: UIS reuse across instruction lifetimes. α represents the time for adoption of new instructions, β is the time to outdate instructions and Δ is the minimum instruction lifetime.

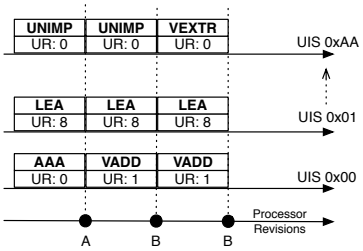


Figure 7: The orthogonality of UIS and URs. We show three UIS through time. From UR A to B, UIS 0x00 is changed from AAA to VADD, requiring a new revision (B). Afterwards, UIS 0xAAA goes from Unimplemented to VEXTR, which does not require a new UR value nor changes revision.

$$RV_x = \{UR_0, \ldots, UR_N\}$$

The *Trap Mask (TM)*, then, is defined as a vector produced by the element-wise $\neq$ operator between two RVs. In this way, $TM_{x,y}$ gives the list of all instructions that must be emulated when running SR_x in PR_y. Thus, for PR_A and PR_B in Figure 7 we have $RV_A = \{0,8,\ldots,0\}$ and $RV_B = \{1,8,\ldots,0\}$. $TM_{A,B}$ is calculated when a program with SR_A is executed on PR_B:

$$\begin{aligned} TM_{A,B} &= RV_A \otimes RV_B \\ &= \{0,8,\ldots,0\} \otimes \{1,8,\ldots,0\} \\ TM_{A,B} &= \{1,0,\ldots,0\} \end{aligned}$$

The $TM_{A,B}$ indicates that UIS $0x00$ needs emulation and a trap is generated. The software handling the trap is then responsible for providing proper emulation routines. By carefully choosing the instructions to retire, manufactures can keep this emulation overhead low.

Merging Revisions. The operator $\oplus$ is defined as a element-wise *OR* between two revision vectors. Given two sequential but not successive PRs, PR_a and PR_e, the operator *Pack* is defined in Equation 1.

$$\begin{aligned} Pack(PR_a, PR_e) &= TM_{a,e} \oplus TM_{b,e} \oplus \ldots \oplus TM_{d,e} \\ &= \{ae_0 \oplus be_0 \oplus \ldots \oplus de_0, \ldots\} \end{aligned} \quad (1)$$

5. Implementation Aspects of SHRINK

This section discusses the implications of a possible implementation of SHRINK's recycling mechanism. We use the 32-bit x86 ISA as our case study owing to its long history. Seeking an efficient hardware implementation, we restrict ourselves to backward compatibility[1].

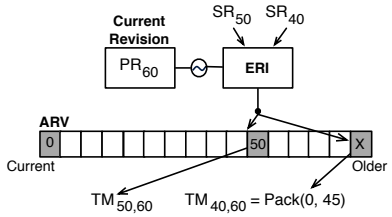


Figure 8: General trap mechanism using the ARV. A software with SR_{50} yields an $ERI = 10$ indexing into $ARV[10]$, which generates $TM_{50,60}$. A SR_{40} has the $ERI = 15$, which indexes the last ARV position, meaning that the resulting trap mask is $TM_{40,60} = Pack(PR_0, PR_{45})$.

5.1. General Mechanism

Active Revision Vector (ARV). For a given PR_i, all SR_j with $j < i$ needs emulation for at least a single instruction. A good strategy is to efficiently support the most recent SRs, while older ones are supported at a higher cost. Thus, we use an *Effective Revision Index (ERI)*, a 4-bit index that encodes all supported revisions: the current revision, 14 previous ones, and a special index that represents all the others collapsed. An ERI is an index into the *Active Revision Vector*, where each element contains a Trap Mask (TM) that answers which UISes to emulate. Given an ARV element, we select a TM that the processor front-end can use to find trap candidates. ERI values for older revisions map into the same ARV element whose TM is calculated using the *Pack* operator (Figure 8).

5.2. Hardware

The hardware design of our processor should keep track of the current ERI, calculated from the software revision currently in execution. There are a few ways to implement this, ranging from the creation of a new instruction – informing the processor the ERI of the next instructions (instruction granularity) – to a modified virtual memory architecture that embeds the ERI information in each memory page (page granularity). The instruction granularity implies the usage of a custom instruction to signal the processor that new code is about to run, like mode switch instructions in some ISAs. To avoid having to add glue code between library calls that potentially change the software version, we adopt the virtual memory approach.

Page table extension. We extend page table entries to include a 4-bit ERI, which entails three modifications. First, page table entries must reserve bits for the ERI (e.g., reserve currently unused entry bits, such as 62–52). Second, instruction TLB entries must accommodate the ERI. We estimate that this expansion would require 6% extra TLB capacity in a computer with 40-bit physical addresses. Third, introducing

[1] We leave forward compatibility as future work. Using forward compatibility, old processors could emulate instructions from newer ones.

ERI requires that the linkers request the loader to map codes with different SRs to different pages.

Processor Front-end. Alongside the decoder, additional hardware must identify instructions that require emulation and issue a trap. However, the size of this extra circuitry can be made quite small if compared to the area saved in the decoder by removing instructions. If the trap checking is integrated into the decoder, it will share its inputs (the UIS) and will only need to output an additional 4 bits indicating the last ERI for which emulation is required. Afterwards, a simple comparer decides to trap if the current ERI is equal to or less than this number. Conversely, a full decoder needs to output several bits corresponding to the μops that an x86 instruction generates. Thus, a significant overall advantage comes from reducing the μROM by removing instruction implementations from hardware and transferring them to software. Since traps are only generated at commit stage, SHRINK can check instructions out of the critical path.

Verification. Any new piece of hardware increases the verification effort. It is not different with instruction recycling, but the mechanism itself can significantly reduce chip verification efforts by reducing instructions implemented in hardware.

ISA Domain Specialization. By carefully removing instructions in newer revisions, it is possible to drastically reduce the ISA size and allow it to enter market segments where the x86 complexity previously was seen as a negative aspect, such as the embedded and extreme low-power market segments. Alternatively, ISA extensions could be created for specific domains without impacting other processors in the family. This is already explored even with no recycling, e.g., Intel Xeon Phi, which supports a unique 512-bit AVX extension without supporting previous extensions, spurring discussion about its backwards compatibility. With our mechanism, Intel would be free to explore lighter ISAs without compatibility issues.

5.3. Software

The software portion is most notably composed of the emulation code layer that supports the execution of removed instructions, but is not restricted to it. Linkers, loaders and the operating system call API need to be aware of code versions in order to be able to correctly fill out our modified page table entry structures. The SR should be annotated in the file, e.g. a field in the ELF file header. To keep backward compatibility, all non-annotated files should be considered as SR_0.

Linker. Must keep track of the code version of each linked module used to build the final executable. If there is a SR mismatch between two different modules to be linked together into the final executable, the linker must allocate them to different pages, each one augmented with its own SR. No modules with different SR can be put together into the same page unless the linker can prove that they only use compatible UIS. To ease this task, intermediary object files should bear this distinction as well, and assemblers should recognize a new directive that specifies the module's SR.

Operating system loader. Recognizes the SR information for each loadable segment present in the binary envelope (PE, ELF, etc.) and allocates memory pages whose page table entries are marked with the corresponding ERI for this SR. To this extent, memory page allocation system calls must also be expanded with the SR parameter because the application may wish to allocate pages that contain code rather than data.

Emulation routines. Mimic the behavior of removed instructions, with important restrictions. They must avoid using outdated instructions; they cannot use removed instructions; and they cannot implement the behavior of instructions that change internal processor registers, including instructions bearing special characteristics, like atomicity, which cannot be reproduced via software. For example, the `clflush` instruction (which flushes cache lines) cannot be emulated nor replaced. However, the number of such instructions is small.

Another important issue is to determine where the emulation routines should reside. We could place the emulation code either in the machine firmware or in emulation drivers that are loaded as early as possible by operating systems. The first alternative has the advantage of being able to emulate old instructions from the operating system itself, since the routines do not rely on the operating system to be loaded, but are already present in the firmware. The second alternative is to use the operating system to handle the emulation traps. Without loss of generality, we will focus on this method for the rest of the paper. After receiving the emulation trap, the operating system decodes the instruction, checks for the software revision, and dispatches the execution to the instruction implementation. We expect that every operating system uses the same implementation for the same instruction and this implementation should be made available by the processor manufacturers to guarantee compatibility.

On the other hand, really old operating systems would not be able to run on new processors because they may use removed instructions. We do not expect this to be a problem because neither are current platforms able to run old operating systems for a similar reason: they lack drivers to support modern hardware. As we pointed out before, modern hardware manufacturers, such as Asus, Gigabyte and Evga, do not provide drivers for old operating systems. However, an old operating system released just a few years before a new processor should work because during the *outdating* step it already avoided the use of the instructions just removed.

Note that emulating instructions in software in the long term may lead to more emulated instructions than real ones. That is not a problem since removed instructions will be rarely used.

Security implications. By changing the ERI/SR of one page or file, we may change the behavior of the software. However, changing a memory page or tampering a file may already require higher privileges on a machine, so that the

user could harm the system in other ways. Future work should investigate the security implication of our approach.

Limitations. One limitation of this mechanism is where to store the emulation libraries and how to distribute them. We have discussed alternatives through firmware and operating system and both requires a good communication channel to provide implementations of recycled instructions. By relying on software vendors, it is also possible to them to deliberately not support one specific revision on their systems, restricting the software able to run in that OS. Last, there may appear intellectual property issues when moving one implementation from hardware to software.

6. Evaluation

In this section, we first evaluate the impact of SHRINK on the area, the critical path and the power usage of the instruction decoder. Later we present a case study of the impact on code size and instruction cache (I-cache) misses after re-encoding frequent AVX instructions with smaller, recycled UISes. Finally, we investigate the performance impact of emulating retired instructions.

6.1. Impact on the Instruction Decoder

In modern x86 processors, the instruction decoder and the microcode ROM occupy a non-negligible area of each core. For example, on the Intel Xeon E5-2600 those components occupy approximately 17% of the area of each core [26], whereas on the AMD Jaguar, that number is close to 12% [25].

To estimate the impact of ISA aging on the instruction decoder and to assess the benefits of SHRINK, we synthesized and compared instruction decoders for the full x86 instruction set, across various generations of x86 processor releases. The baseline decoders were generated based on the original ISAs, as they were released. We applied the SHRINK methodology on the baseline decoders to reduce their ISA complexity and assess the impact of the simplified decoders on the critical path, area, and power consumption for all decoder designs. Finally, we also consider a theoretical Naive approach for ISA simplification, which merely eliminates retired instructions from the ISA and violates backward compatibility.

Synthesis and design. We synthesize our decoder circuitry using the Cadence's Encounter RTL compiler with the FreePDK design kit [29]. For easier comparison, we used the smallest lithography process available in FreePDK (45nm technology) and clocked all decoders at 500MHz.

In our decoder design, we take into account the number of generalized and restricted decoders in each processor release [21, 27], where only generalized decoders are able to decode and translate complex instructions (>4μops). To classify an instruction as simple or complex, we follow Fog's method [10]. Our evaluation is split into two parts, we first evaluate the decoder logic and later the microcode ROM.

Instruction removal policy. We select instructions for removal from the ISA using the methodology in Section 4.1. Specifically, we apply different values of α, β, and Δ on the dynamic profiles of all virtual machines in Table 1, organized by year – for each year, we aggregated the latest available Windows and Linux versions. We evaluate the impact of the resulting instruction removal policies on the decoder circuitry.

Evaluation. Figure 9 shows the expected execution time overhead when we emulate all removed instructions. The Figure used $\beta = 2$, as the result for $\beta = 4$ was very similar. From the figure, the minimum Δ with afforded overhead of 10% is 8, meaning that we cannot remove instructions created in less than 8 years without incurring high overhead. Notice that, by adopting our approach and using the Outdating period, users should not face this overhead because software designers should gracefully remove the outdated instructions, and for old software, we can count on performance scaling better than 10% in a 2-year window to minimize the overhead. Indeed, if we consider SPEC benchmark reports, the smallest gain in a 8-year window is 6 times.

Figure 10 shows the number of removed instructions in time, with respect to the total number of instructions in each release. Notice that for smaller Δ we have more instructions removed since we are not giving the adequate adoption time. Again, $\Delta = 8$ is a good choice here, removing from 30% to 40% of all instructions.

Figure 11 evaluates SHRINK and Naive decoders in terms of critical path, area and power consumption. The critical path improves by up to 50% (average of 23%), and SHRINK performs better than Naive mostly because it reuses old UIS instead of creating new ones for every new instruction. The outlier in 2007 is due to the high number of new instructions in SSE 4.1 using longer encoding. The decoder area was drastically reduced by up to 75% (average of 48%). This huge area improvement was due to the fact that the original decoder has to hold all instructions while SHRINK not only removes unused UIS but also reuse them to encode the new ones. Finally, the power consumption was reduced by up to 70% (average of 49%). Again, the huge improvement stems from reusing old instructions to encode new UISes, rather than using the new (and longer) UISes.

In addition to evaluating the decoder logic, we approximate the impact of our approach on the microcode ROM by assessing the number of complex instructions removed in each decoder. We found that when the removal policy is activated, a substantial amount of those instructions are quickly removed from the ISA due to lack of use. Using $\beta = 2$, $\Delta = 4$ removed 51% of complex instructions, $\Delta = 5-6$ removed 47%, $\Delta = 8-12$ removed 43%, and $\Delta = 14$ removed 41%. For $\Delta = 8-12$ the 43% of complex instructions represent 32% of the micro-operations stored in the microcode ROM. Because the size of the ROM depends directly on the amount of micro-operations it holds, reducing its storage requirements should translate into proportional area, power, and latency reductions.

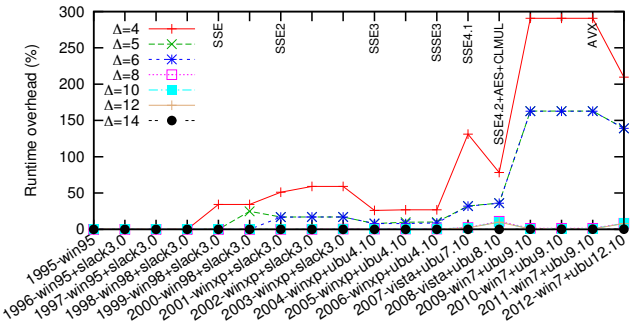


Figure 9: SHRINK's runtime overhead for different values of Δ

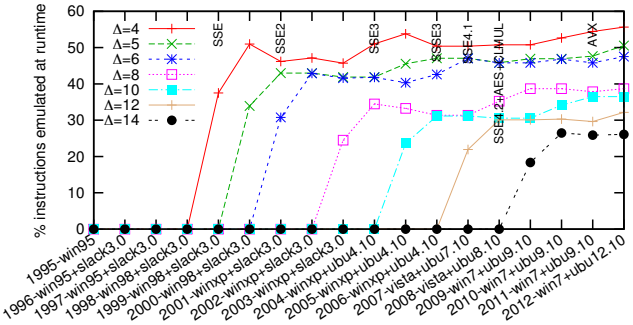


Figure 10: Instructions removed per processor release, normalized to the total number of instructions in the release

6.2. Case Study: Re-encoding AVX using SHRINK

Vector Extensions. The IA-32 ISA recently had almost all extensions focused on adding vector instructions that explore data parallelism. The first extensions to address floating-point calculations were 8087 and 80387, but their operand addressing is stack-based, a rather old and inconvenient addressing method for modern compilers. Newer vector instructions, starting with the MMX extension, have register operands, which allows the compiler to easily control register usage with established register allocation algorithms and are also able to perform multiple floating-point calculations at the same cycle. For these reasons, vector instructions naturally supersede the old x87 instructions.

In Figure 12, we show the share that specific IA-32 extensions take into the total dynamic frequency count of each workload, starting at 95%. For example, the chart shows that over 4% of total instructions executed in our Windows 7 workload were extensions instructions. We conclude that IA-x87 extensions did not stop being used over time, albeit being functionally superseded by newer extensions. In fact, it has increased its participation on the total instruction count over time in Windows systems. For this reason, the ISA is forced to be redundant.

Figure 13 shows that 7 SPEC CFP 2006 programs using vector extensions yields larger executable binaries than the x87 ones. In this analysis, we compiled the programs with *gcc* version *4.7* using the *-O2* optimization flag and the *-march=corei7-avx* architecture tuning. To generate x87 instructions, we used set *-mfpmath=387*, whereas to generate the SSE instructions, we set *-mfpmath=sse*. Finally we enabled the *-mavx* flag when testing the AVX encoding for SSE instructions. No vectorization optimizations were used.

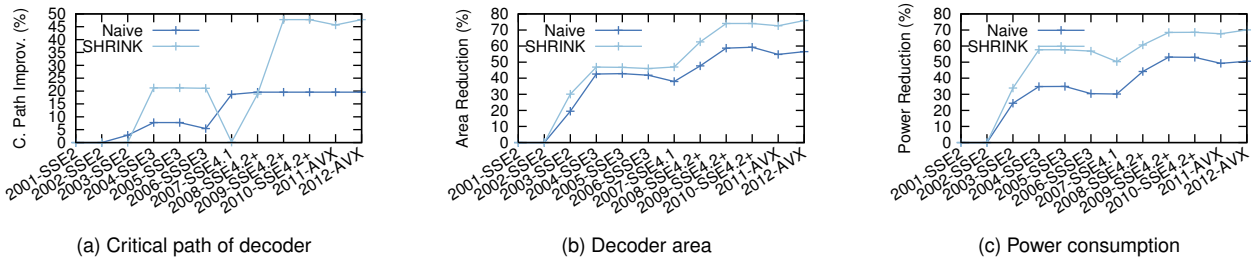
Recalling Figure 2, we know that vector extension instructions may have 1 to 2 extra bytes for UIS as compared to old IA-x87 extension. The increased code size may explain why compilers still generate old IA-x87 instructions in favor of newer SSE or AVX instructions when there is no data parallelism. It is also known that, to this date, IA-x87 floating-point instructions are still the default in widely used compilers, like gcc. The outdating mechanism in this paper allows for a smooth transition between extensions, being ideal for such a difficult scenario as the x87.

AVX Re-encoding. We analyze the improvement on total program code size and on instruction cache misses when the most frequent AVX instructions replaced the UISes of shorter, retired instructions. In particular, we considered 2 re-encoding scenarios. Each scenario has the form `AVX-(n,m)` where *n* and *m* are the number of re-encoded 1-byte and 2-byte UISes, respectively. For example, in the `AVX-(5,6)` scenario, we use 5 1-byte and 6 2-byte recycled UISes for AVX re-encoding.

Figure 14 presents the total code size of SPECfp programs in each scenario, using their old AVX version as the baseline for comparison. The chart also shows the size of a version of the program compiled using the old IA-x87 extension for floating point arithmetic. The original AVX version of these programs is, on average, 6% bigger than their IA-x87 version in total code size. On small programs that have a higher percentage of floating point instructions, as in the `470.lbm`, the total code size can be 15% bigger. However, vectorization support is not available in the old IA-x87. The goal of this re-encoding is to offer modern vectorization support with the same code compaction achieved with the first instructions introduced in the IA-32 ISA.

Even though AVX re-encoding may seem expensive in face of the high number of instructions in this extension, in practice, it is enough to encode 5 of the most frequent AVX instructions using 1-byte UISes and 6 of the remaining most frequent AVX instructions using 2-byte UISes – the AVX-(5,6) scenario – to generate SPECfp AVX executables, on average, 5.3% smaller than their old AVX versions, almost the code compaction achieved if the user gives up modern vectorization support and uses the old IA-x87 extension for floating point arithmetic. If there is no 1-byte UIS available to encode the most frequent AVX instruction, the `AVX-(0,30)` scenario shows that even if there are 30 2-byte UISes, the total code size is, on average, only 3.2% smaller than the old AVX version.

The most important benefit of re-encoding is its effects on the instruction cache. Considering the `AVX-(5,6)` scenario, we used the dinero cache simulator [6] to measure the instruction cache miss reduction impact of re-encoded AVX programs from the SPECfp benchmark. Based on a recent *Ivy Bridge*



(a) Critical path of decoder (b) Decoder area (c) Power consumption

Figure 11: Critical path, area, and total power gains for Naive and SHRINK relative to our baseline decoder

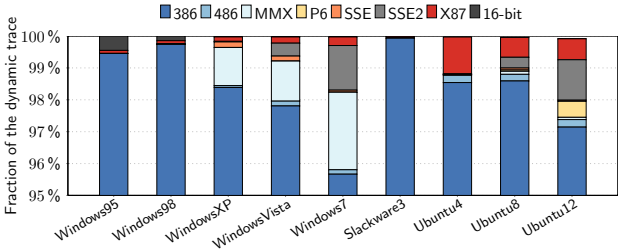


Figure 12: Percentage of extension instructions executed in Windows and Linux dynamic traces.

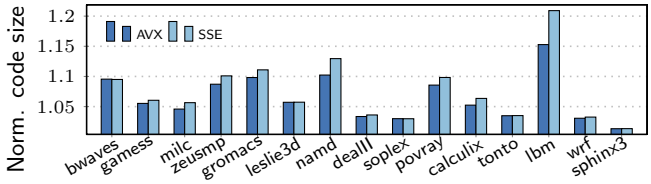


Figure 13: Code size of SPEC CFP 2006 using SSE and AVX, baseline is IA-x87 code

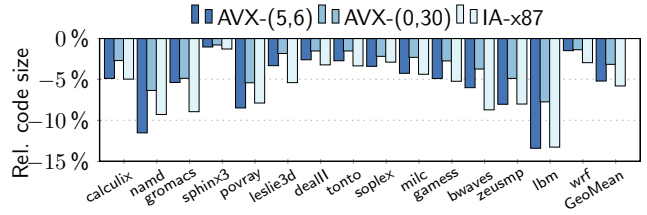


Figure 14: Code size of SPEC CFP 2006 using different AVX re-encodings relative to the original AVX encoding

Core i7 cache configuration, we set dinero to use a 32K instruction L1 cache, without L2, in two different configurations: 4-way and 8-way associative.

Note that our I-cache simulation results do not necessarily imply direct performance gains. However, as previous research has demonstrated, the I-cache hit ratio is crucial for many modern applications, such as Online Transaction Processing applications [2], web servers, and Cloud computing workloads [7]. Figure 15 presents our results for the reduction of instruction cache misses when long AVX UISes are replaced with shorter reused UISes. Since these frequent instructions now uses less space and the cache can hold more instructions, we can expect a cache performance increase. This effect is

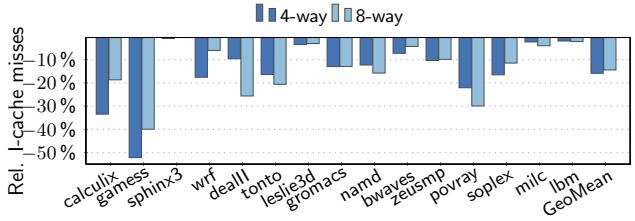


Figure 15: I-cache misses of SPEC CFP 2006 using short AVX encodings relative to the original AVX

shown by the graph, which compares the cache misses of the new encoding against the original, long, encoding, whose cache misses were used as our baseline for comparison. For example, gamess suffers 53% less cache misses if the x86 AVX instruction set uses a more compact way to encode its UISes, representing our best case for cache miss reduction. In the worst case, sphynx had only 0.4% less cache misses using the 8-way cache configuration. Even though the cache misses reduction for the 8-way cache were more modest, in some special cases, the 8-way cache had even greater reduction than the 4-way cache, showing us that the cache effects, although difficult to predict, are always beneficial. On average, there was 16% less cache misses using the 4-way cache and 15% less cache misses using the 8-way cache.

6.3. Assessing SHRINK's Emulation Penalty

In this section, we analyze to what extent deprecated instruction emulation hinders software performance, showing the impact on performance of incrementally moving x86 instructions from hardware to software emulation. We used an analytical model that is based on our dynamic execution traces of several operating system stacks and of SPEC CPU2006. This model estimates performance by tallying executed instructions in a trace. For each occurrence of a retired instruction, we add an emulation penalty, in number of instructions. We did not consider multimedia extensions because it would incorrectly report them as a large percentage of unused instructions, when in fact they are still in adoption by recent software.

Figure 16 shows the result of our analysis. When admitting a maximum of 5% of performance overhead, the graph shows the maximum percentage of the x86 ISA that can be emulated and also shows that with more efficient routines we are able to retire more instructions.

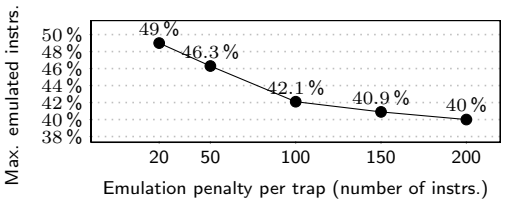


Figure 16: Maximum ratio of emulated instructions, assuming different emulation penalties and a maximum performance degradation of 5%.

For a realistic upper bound for the emulation penalty, we implemented a Linux kernel module that installs an x86 interrupt vector number 3 handler, usually used as a software debugging trap, to emulate the behavior of selected x86 instructions. We also patched the SPEC CPU2006 binaries to replace the first byte of the selected instructions with the `INT3` opcode. The selection includes variants of `mov` that store immediate values in a register and variants of `cmp` that perform comparisons between the `AL` register and an 8-bit immediate value.

Our emulation penalty experiment works as follows. When running a SPEC program, if the next instruction is either `mov` or `cmp`, the processor generates a trap, since the instruction was previously patched with the `INT3` opcode. The kernel module then queries a shadow image of the binary to retrieve the original instruction. We used Bochs to decode the instructions, jump to and execute emulation routines and return to the next instruction in the program. We found that the average emulation time per trapped instruction was 160 processor cycles in SPEC CPU2006 programs even using Bochs full-fledged decoder instead of an optimized one. Therefore we consider 160 cycles to be an upper bound for simple x86 data processing instructions. We conclude that at least 40% of the x86 ISA, even after excluding multimedia extensions, could be emulated with minor performance overhead in the analyzed execution traces.

7. Related Work

Limited backward compatibility. Some processor manufacturers admit some loss of backward compatibility, in trade for cost-performance optimizations. For example, the IBM POWER8 processor deletes 35 POWER1 and 8 POWER2 instructions from its current ISA [15], and the new 64-bit ARMv8 deprecates several ARMv7 instructions [12]. However, no manufacturer leverages instruction emulation to maintain backward compatibility, as we do.

Recent studies argue that RISC and CISC ISAs are nowadays just design point alternatives [5] while other studies argue that we should take advantage of ISA diversity to improve system characteristics [30]. SHRINK does not try to compare ISAs but rather to clear the path for future improvements without losing backward compatibility.

Flexible ISAs. Barrantes *et al.* [4] and Kc *et al.* [22] studied ISA randomization to avoid security attacks, whereas Barat *et al.* [3] studied flexible hardware-software interfaces, as opposed to a fixed ISA in the context of reconfigurable instruction set processors. Such systems share the same purpose of designing functionality-rich ISA so that application code can be densely encoded, thereby reducing the total number of executed instructions and increasing efficiency. However, these solutions focus on reconfigurable hardware [11, 20] that adapts to software needs rather than redesigning the ISA.

Interestingly, the Altera Nios [1] (used in Altera FPGAs) is capable of removing some instructions from hardware and emulates them in software, sharing these similarities with our work. However, the system designer may prefer to allocate the FPGA hardware resources to IPs that increases the overall chip functionality, rather than instantiating a resource-expensive processor. Similarly, application-specific instruction sets (ASIPs) [14, 19] allow extreme processor customization by removing and adding new instructions tied to the application requirements. Conversely, our approach is not limited to FPGA-based soft processors, and it enables the reorganization of the UIS space using version codes for deprecated software, leaving space for new instructions at the same time.

8. Conclusions

In this work we studied the effects of instructions set architectures (ISA) aging and proposed a new mechanism to enable smooth ISA evolution and renovation without breaking compatibility with legacy code.

We first performed an extensive chronological analysis of several Linux and Windows x86 applications to investigate which instructions are frequently used and how they stop being used over time. Then, we showed that these unused instructions might worsen the size, power and performance of x86 instructions decoders. Finally, we proposed and evaluated SHRINK, a mechanism to remove old instructions without compromising backwards compatibility.

Our experimental results indicate that SHRINK allows us to remove several instructions from the x86 ISA and improve the instruction decoder critical path, area, and power consumption, respectively, by 23%, 48%, and 49%, on average. The reuse of smaller opcodes on frequently used instructions also significantly improves code density, reducing cache misses and improving the memory bandwidth. Furthermore, we found that the emulation overhead on legacy software is fairly low: less than 5% when emulating up to 40% of the x86 ISA.

9. Acknowledgements

We thank Lois Orosa, Emilio Francesquini, Alexandro Baldassin, and the anonymous reviewers for their suggestions. This work was supported by FAPESP (grants 2009/02270-0, 2011/09630-1, 2012/50732-5, 2013/05257-0, and 2013/08293-7), CNPq (grants 308517/2014-8 and 449996/2014-0), and CAPES (grant 2966/2014).

References

[1] Altera, *Nios II Processor Reference Handbook*, 2011.

[2] I. Atta *et al.*, "STREX: Boosting Instruction Cache Reuse in OLTP Workloads Through Stratified Transaction Execution," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, 2013.

[3] F. Barat, R. Lauwereins, and G. Deconinck, "Reconfigurable Instruction Set Processors from a Hardware/Software Perspective," *IEEE Trans. Softw. Eng.*, vol. 28, no. 9, Sep. 2002.

[4] E. G. Barrantes *et al.*, "Randomized Instruction Set Emulation," *ACM Trans. Inf. Syst. Secur.*, vol. 8, no. 1, Feb. 2005.

[5] E. Blem, J. Menon, and K. Sankaralingam, "Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures," in *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, Feb 2013, pp. 1–12.

[6] J. Edler and M. Hill, "Dinero IV Trace-Driven Uniprocessor Cache Simulator," 2003.

[7] M. Ferdman *et al.*, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[8] A. Fog, "Future instruction set: AVX-512," http://www.agner.org/optimize/blog/read.php?i=288, accessed May 2014.

[9] A. Fog, *Instructions for Objconv*, 2011, version 2.11.

[10] A. Fog, "Lists of Instruction Latencies, Throughputs and Micro-Operation Breakdowns for Intel, AMD and VIA CPUs," 2014, www.agner.org/optimize/instruction_tables.pdf.

[11] C. Galuzzi and K. Bertels, "The Instruction-Set Extension Problem: A Survey," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 4, no. 2, May 2011.

[12] T. R. Halfhill, "ARM's 64-Bit Makeover," The Linley Group Newsletters, December 2012.

[13] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Comput. Archit. News*, vol. 34, no. 4, Sep. 2006.

[14] I.-J. Huang and A. M. Despain, "Synthesis of Application Specific Instruction Sets," *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, vol. 14, no. 6, Nov. 1995.

[15] *Power ISA*, Version 2.07 ed., IBM, 2013.

[16] *IA-32 Intel Architecture Software Developer's Manual*, Volume 2: Instruction Set Reference ed., Intel Corporation.

[17] Intel Corporation, "Intel AVX: NewFrontiers in Performance Improvements and Energy Efficiency," 2008, white paper.

[18] Intel Corporation, "Haswell New Instruction Descriptions Now Available," http://software.intel.com/en-us/blogs/2011/06/13/haswell-new-instruction-descriptions-now-available, 2012, accessed May 2014.

[19] M. Jain, M. Balakrishnan, and A. Kumar, "ASIP Design Methodologies: Survey and Issues," in *VLSI Design, 2001. Fourteenth International Conference on*, 2001.

[20] L. Jówiak, N. Nedjah, and M. Figueroa, "Modern Development Methods and Tools for Embedded Reconfigurable Systems: A Survey," *Integr. VLSI J.*, vol. 43, no. 1, Jan. 2010.

[21] D. Kanter, "Intel's Haswell CPU Microarchitecture," Real World Technologies, November 2012, http://www.realworldtech.com/haswell-cpu/2/, accessed May 2014.

[22] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks With Instruction-Set Randomization," in *CCS '03*, 2003.

[23] A. Kilmovitski, "Using SSE and SSE2: Misconceptions and Reality," Intel Developer Update Magazine, 2001.

[24] K. P. Lawton, "Bochs: A Portable PC Emulator for Unix/X," *Linux J.*, vol. 1996, no. 29es, Sep. 1996.

[25] T. P. Morgan, "AMD to Double up Cores With Jaguars And Maybe Finally a Cat Server Variant," http://www.theregister.co.uk/2012/08/29/amd_jaguar_core_design/, 2012, accesses May 2014.

[26] T. P. Morgan, "Intel Plugs Both Your Sockets With 'Jaketown' Xeon E5-2600s," http://www.theregister.co.uk/2012/03/06/intel_xeon_2600_server_chip_launch/, 2012, accessed May 2014.

[27] J. Shen, *Modern Processor Design: Fundamentals of Superscalar Processors*, ser. McGraw-Hill Series in Electrical and Computer Engineering. McGraw-Hill Companies,Incorporated, 2004.

[28] A. Shrivastava *et al.*, "Compilation Framework for Code Size Reduction Using Reduced Bit-Width ISAs (rISAs)," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 1, Jan. 2006.

[29] J. E. Stine *et al.*, "FreePDK: An Open-Source Variation-Aware Design Kit," in *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education*, 2007.

[30] A. Venkat and D. M. Tullsen, "Harnessing ISA Diversity: Design of a heterogeneous-ISA Chip Multiprocessor," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ser. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 121–132. Available: http://dl.acm.org/citation.cfm?id=2665671.2665692

[31] V. Weaver and S. McKee, "Code density concerns for new architectures," in *Computer Design, 2009. ICCD 2009. IEEE International Conference on*, Oct 2009.