What about a two-dimensional virtual address space?

Pierre Michaud

What about a two-dimensional virtual address space?

Pierre Michaud

RESEARCH REPORT
N° 9563
December 2024
Project-Teams PACAP

ISSN 0249-6399 ISRN INRIA/RR--9563--FR+ENG

What about a two-dimensional virtual address space?

Pierre Michaud*

Project-Teams PACAP

Research Report n° 9563 — December 2024 — 27 pages

Abstract: The virtual address space offered by existing instruction set architectures (ISA) is generally one-dimensional. That is, a virtual address is represented by a single number. We consider the possibility of a two-dimensional (2D) address space where a virtual address consists of two independent numbers, an X coordinate and a Y coordinate. We propose and describe a hypothetical ISA, called XYA, offering a 2D virtual address space. We explain how a XYA CPU would differ from a conventional CPU and we propose a C-like language for programming a XYA machine. We identify a set of XYA features that conventional ISAs do not have and we explain how performance-aware programmers can exploit these features.

Key-words: instruction-set architecture, virtual memory, address translation, microarchitecture, TLB, cache, performance-aware programming, pointer, array

* Inria, Univ Rennes, CNRS, IRISA

RESEARCH CENTRE
Centre Inria de l'Université de Rennes

Campus universitaire de Beaulieu
Avenue du Général Leclerc
35042 Rennes Cedex

Proposition d'architecture généraliste ayant un espace d'adresses virtuelles bidimensionnel

Résumé : Les jeux d'instructions existants ont un espace d'adresses virtuelles unidimensionnel, une adresse virtuelle étant représentée par un nombre entier. Dans ce document, nous considérons la possibilité d'un espace d'adresses virtuelles à deux dimensions, une adresse virtuelle consistant en deux nombres indépendants, une coordonnée X et une coordonnée Y. Nous proposons un jeu d'instructions hypothétique, nommé XYA, ayant un espace d'adresses virtuelles bidimensionnel. Ce document met en évidence les principales différences entre un processeur XYA et un processeur classique. Nous proposons un langage de programmation ressemblant au langage C et permettant d'exploiter la mémoire bidimensionnelle d'une machine XYA. Enfin, nous soulignons certaines caractéristiques de XYA, différentes des jeux d'instructions classiques, et montrons comment les programmeurs peuvent les exploiter pour obtenir des programmes s'exécutant plus rapidement.

Mots-clés : jeu d'instructions, architecture, mémoire virtuelle, traduction d'adresses, microarchitecture, TLB, cache, programmation pour la performance, pointeur, tableau

Contents

1 Introduction

The instruction set architecture (ISA) is the interface between the software and the hardware. A general-purpose ISA provides programming primitives that can be used by compilers and can be implemented efficiently in hardware.

Modern ISAs distinguish the memory addresses seen by the programmer, aka *virtual* addresses, from the *physical* addresses that the hardware uses. The hardware and the operating system collaborate to translate virtual addresses into physical addresses, without the programmer's intervention. Such virtual memory provides several advantages. In particular, it makes programming easier and programs portable.

Most ISAs today use paging-based address translation: the address space is divided into *pages* whose size is a power of two (e.g., 4 KB) and the virtual page address is translated into a physical page (aka *frame*) address [8, 23, 42]. The virtual address space of current ISAs is *linear*, i.e., one-dimensional (1D): a virtual address is a scalar number, and a page is a set of contiguous virtual addresses.

Programmers for whom performance (i.e., program execution speed) is not a problem may ignore the linearity of the address space, which can be hidden inside library functions or behind convenient programming abstractions. However, the linearity of the address space is apparent to performance-aware programmers, who need a basic understanding of how TLBs[1] and caches work to organize data in memory and access it in a way that gives good performance.

Linearity seems a somewhat arbitrary choice. The data manipulated by programs are often inherently multidimensional. Multidimensional data can be mapped into a one-dimensional address space, however such data flattening has a programmability and performance cost.

It is unlikely that we will be stuck forever with the ISAs that are prevalent today. At some point in the future, new ISAs will likely be created. It is therefore a legitimate research question to ask whether there would be some advantages in departing from the conventional linear virtual memory.

In particular, we consider the possibility of a two-dimensional (2D) virtual address space that can be made visible to programmers via a C-like programming language. Our intuition is that adding an extra dimension to the address space offers new degrees of freedom making performance-aware programming easier.

We propose a possible 2D virtual memory architecture called XYA, and a C-like programming language called XYC for programming with XYA. Finally, we identify some advantages of XYA for the programmer.

2 XYA: a two-dimensional virtual memory architecture

This section describes a 2D virtual memory architecture called XYA (for XY architecture). We assume definite numerical values for important XYA parameters, to make the description as concrete as possible. Nevertheless, XYA is a sketch, not a completely specified ISA. We focus the description on what differentiates XYA from a conventional ISA (mainly addressing modes and address translation). To ease the discussion, we assume a RISC-like ISA.

We assume that a *virtual* address consists of a pair of coordinates (X, Y). XYA is a 2x64-bit architecture. That is, each coordinate is 64-bit long, and the whole virtual address is 128-bit long. Each distinct address points to a distinct byte in memory. We assume conventional, one-dimensional physical addresses whose width does not exceed 64 bits.

[1] Translation Lookaside Buffers

term	definition
rectangle	$[X_A, X_B] \times [Y_A, Y_B] =$ $\{(X, Y) : X \in [X_A, X_B]$ and $Y \in [Y_A, Y_B]\}$
rectangle address	(X_A, Y_A)
width	$X_B - X_A + 1$ (dimensionless)
height	$Y_B - Y_A + 1$ (bytes)
area	width times height (bytes)
aspect ratio	height divided by width
x-aligned	Y_A is a multiple of $Y_B - Y_A + 1$
y-aligned	X_A is a multiple of $X_B - X_A + 1$
aligned	simultaneously x-aligned and y-aligned
ground	addresses whose Y coordinate is zero
grounded	whose address has a null Y coordinate
pile	rectangle of width 1
silo	silo X is the unit-width rectangle $[X, X] \times [0, 2^{64} - 1]$
1D software	software using a single silo
2D software	software using multiple silos

Table 1: Glossary.

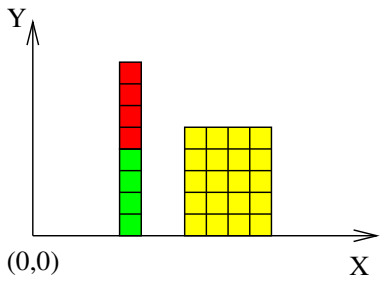

Figure 1: The green and red piles are in the same silo and have 4-byte area each. The yellow rectangle has width 4, height 5 bytes, and area 20 bytes. The green pile and the yellow rectangle are grounded.

While this study is mostly concerned with the possibilities offered by a 2D address space, a practical 2D virtual memory architecture should allow legacy software to run with minimal modifications. To this end, the 2D address space of XYA can be used as a collection of independent 64-bit linear address spaces, like a *segmented* memory [42]: the X coordinate represents the segment number, and the Y coordinate is the linear address within the segment.

2.1 Terminology

Let (X_A, Y_A) and (X_B, Y_B) be two virtual addresses such that $X_A \leq X_B$ and $Y_A \leq Y_B$. The *rectangle* $[X_A, X_B] \times [Y_A, Y_B]$ is the set of addresses (X, Y) such that $X \in [X_A, X_B]$ and $Y \in [Y_A, Y_B]$ with notation $[X_A, X_B]$ for the set $\{X : X \in \mathbb{Z}$ and $X_A \leq X \leq X_B\}$. The rectangle address is (X_A, Y_A), its *width* is $X_B - X_A + 1$, its *height* is $Y_B - Y_A + 1$ bytes, its *area* is the width times the height, and its *aspect ratio* is the height divided by the width. The *ground* is all the addresses with a null Y coordinate. The Y coordinate of the address of a *grounded* rectangle is zero. A *pile* is a unit-width rectangle. A *silo* is a grounded pile of height 2^{64} bytes. See Table 1 for a glossary and Figure 1 for an illustration.

In this paper, *1D software* means that the code and data are confined in a single silo. This is for instance legacy software written for conventional 64-bit architectures. *2D software* is software using multiple silos for its code and/or data.

2.2 Control flow

The sequencing of instructions in conventional architectures is essentially linear, as the program counter (PC) is incremented automatically after each instruction except for branches. This matches the linear control flow of most programming languages. While it might be interesting to question this postulate, we leave meditations on this question to future studies and we assume a linear control flow.

Each instruction occupies a 4-byte aligned pile. The program counter is a virtual address PC = (PCX, PCY). After a non-branch instruction or a not-taken conditional branch, the PC is incremented along the Y direction:

$$(\mathrm{PCX}, \mathrm{PCY}) \longrightarrow (\mathrm{PCX}, \mathrm{PCY} + 4)$$

That is, the natural evolution of the PC is to go upward in the same silo. There are two sorts of jumps: *vertical* and *oblique*. A vertical jump keeps PCX unchanged, that is, the PC stays in the same silo. An oblique jump can jump to another silo. The addressing modes for vertical jumps are similar to those found in conventional ISAs:

vertical relative:	$(\mathrm{PCX}, \mathrm{PCY}) \longrightarrow (\mathrm{PCX}, \mathrm{PCY} + \mathrm{IY} \times 4)$
vertical indirect:	$(\mathrm{PCX}, \mathrm{PCY}) \longrightarrow (\mathrm{PCX}, \mathrm{RY})$

where IY is a signed immediate offset and RY is any integer register. A vertical call is a vertical jump that saves PCY + 4 in an implicit register. Oblique jumps are unconditional and either indirect or semi-indirect:

oblique indirect:	$(\mathrm{PCX}, \mathrm{PCY}) \longrightarrow (\mathrm{RX}, \mathrm{RY})$
oblique semi-indirect:	$(\mathrm{PCX}, \mathrm{PCY}) \longrightarrow (\mathrm{PCX} + \mathrm{IX}, \mathrm{RY})$

where RX and RY are any integer registers and IX is a signed immediate offset. An oblique call is an oblique jump that saves PCX and PCY + 4 in two implicit registers.

Oblique jumps are needed for executing code located in a different silo. We may want to use multiple silos for security reasons. Vertical indirect jumps and oblique semi-indirect jumps provide *siloed addressing*, i.e., the PC is guaranteed to arrive in a definite silo even if the register containing the target Y coordinate has been tampered with. This reduces the attack surface. Siloed addressing can be used to sandbox untrusted code [37, 44].

2.3 Loads and stores

The data accessed by one load/store instruction is laid out in memory as a pile. XYA assumes simple addressing modes for each coordinate, similar to RISC-V.

XYA's PC-relative addressing modes are:

PC-relative vertical:	$(\mathrm{PCX}, \mathrm{PCY} + \mathrm{IY})$
PC-relative semi-direct:	$(\mathrm{PCX} + \mathrm{IX}, \mathrm{IY})$

where IX and IY are signed immediate offsets, except for semi-direct addressing where IY is always positive.

XYA's register-based addressing modes are:

oblique:	$(\mathrm{RX} + \mathrm{IX}, \mathrm{RY} + \mathrm{IY})$
vertical:	$(\mathrm{RX}, \mathrm{RY} + \mathrm{IY})$
horizontal:	$(\mathrm{RX} + \mathrm{IX}, \mathrm{RY})$
semi-direct:	$(\mathrm{RX} + \mathrm{IX}, \mathrm{IY})$

where RX and RY are (any) integer registers and IX and IY are signed immediate offsets, except for semi-direct addressing where IY is always positive.

It should be noted that XYA, like RISC-V, does not provide register+register aka base+index addressing. Without base+index addressing, a loop iterating over several arrays simultaneously must increment a register for each array [45]. However, 2D software can emulate base+index addressing if each array is grounded in a distinct silo, as it is sufficient to increment one register holding the Y coordinate.

XYA also offers addressing modes that are PC-relative in X and register-based in Y:

siloed:	$(\mathrm{PCX} + \mathrm{IX}, \mathrm{RY} + IY)$
1D:	$(\mathrm{PCX}, \mathrm{RY} + IY)$

Siloed addressing can be used to sandbox untrusted code [16, 44, 56]: if a binary software uses siloed addressing only, data accesses are guaranteed to be restricted to definite silos. 1D addressing is useful for 1D software.

2.4 Virtual address space

We assume a paging-based virtual memory with a base page size of 4 KB.[2] That is, a 4 KB *physical* page is a 4 KB contiguous, aligned region of physical memory.

A 4 KB virtual page is an aligned rectangle of area 4 KB. While the virtual page area equals the physical page size, we must also define the aspect ratio of the virtual page. One of the essential features of XYA is that the page aspect ratio is not fixed, it depends on the X coordinate. This provides a new degree of freedom that programmers can exploit when optimizing for performance.

XYA partitions the virtual address space into regions called *books*. The book number B is determined by the leftmost (i.e., most significant) bits of the X coordinate. That is, a book consists of adjacent silos. All the pages in a given book have the same aspect ratio. More precisely:

$$\textit{in book } B, \ \textit{pages have width } 2^B \ \textit{and height } 2^{12-B}.$$

In theory, B can take 13 distinct values. However, as load/store data is laid out as a pile, it is preferable that aligned load/store accesses do not cross page boundaries. In this study, we assume **eight** books $B \in [0, 7]$. This allows load/store data to be up to 32-byte tall without crossing a page boundary (provided the data is aligned).

A conventional linear virtual memory with 4 KB pages leaves the 12 rightmost (i.e., least significant) bits of the virtual address unmodified by the address translation. In other words, the 12 right bits define the physical page offset (**PPO**).

In XYA, the PPO in book B is obtained by concatenating the B right bits of the X coordinate with the $12 - B$ right bits of the Y coordinate (see Figure 2):

$$\mathrm{PPO}_{4\mathrm{K}} = X_{[B-1:0]} \times 2^{12-B} + Y_{[11-B:0]} \tag{1}$$

[2] 4 KB is the base page size in the x86 and RISC-V architectures and one of the possible translation granules in the ARM architecture.

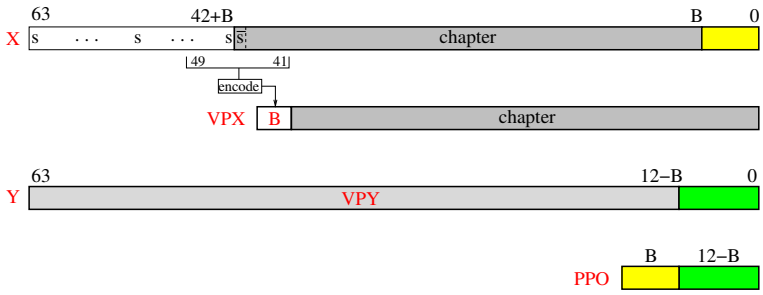

Figure 2: How the book number B, virtual page name (VPX,VPY) and physical page offset $\text{PPO}_{4\text{K}}$ are obtained from the virtual address (X, Y).

where $X_{[L:R]}$ is the number formed by extracting bits of X between positions L and R. In the remaining, notation $X_{[R]}$ means the bit $X_{[R:R]}$.

Each book is divided into a fixed number of *chapters*. Each chapter in book B consists of 2^B adjacent silos, i.e., the chapter width equals the page width. In this study, we assume 2^{42} chapters per book.[3] The *chapter number* is determined by the X coordinate:

$$\text{chapter number} = X_{[41+B:B]}$$

A *legal* X coordinate must have all its bits to the left of bit $X_{[41+B]}$ equal to the complement of $X_{[41+B]}$, where B is a valid book number. That is, assuming a legal X, the book number is determined by the position of the leftmost bit whose value is the complement of bit $X_{[63]}$. With 8 books and 2^{42} chapters, X is legal iff the following condition is true:

$$(X_{[63:49]} = 0 \text{ or } X_{[63:49]} = 2^{15} - 1) \text{ and } X_{[48:41]} \neq X_{[56:49]}$$

The book number B is encoded inside the X coordinate as a *thermometer code* [51]. Given a legal X, B is completely determined by the nine bits $X_{[49:41]}$: we have $B = b - 41$ where b is the greatest number in $[41, 48]$ such that $X_{[b]} \neq X_{[b+1]}$. Note that a legal X in book B, shifted one bit to the left, yields a legal X in book $B + 1$.

The advantage of the thermometer code is that the legal X values consist of two contiguous and symmetrical regions.[4] The "low" region is $X \in [2^{41}, 2^{49} - 1]$ and the "high" region is $X \in [2^{64} - 2^{49}, 2^{64} - 2^{41} - 1]$. Each book has its chapters split equally between the low and high regions. The region can be identified from bit $X_{[63]}$ of the full address or, equivalently, from the leftmost bit of the chapter number. The operating system (OS) can use one region for the kernel and the other region for the user code and data.

The virtual address space of XYA is huge[5] by necessity as XYA must accommodate both 1D and 2D software. A huge virtual address space can be exploited to implement certain protection techniques improving security and reliability [29, 30, 40, 55].

As all books contains the same number of chapters and the chapter width equals the page width 2^B, book $B + 1$ is two times larger than book B. Book 0, the smallest book, is huge nevertheless ($2^{42} \times 2^{64} = 2^{106}$ bytes). In practice, the sole criterion for selecting a book is the page aspect ratio that is deemed the most appropriate. For instance, book 0 is the natural place for 1D software, instructions, and data that are essentially one-dimensional.

[3] The description we provide can easily be adapted to a number of chapters other than 2^{42}.

[4] If we take an X in one region and complement each of its bits, the resulting $\overline{X}$ belongs to the same book in the other region.

[5] There are 255×2^{42} legal X values and 255×2^{106} bytes in total.

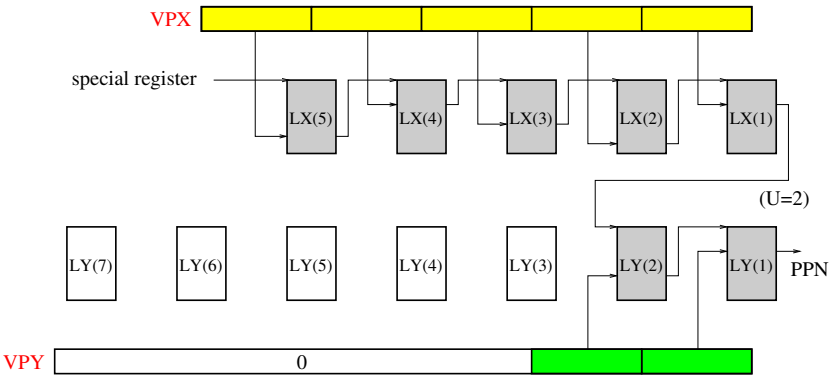

Figure 3: Example of page table walk. In this example, all accesses in this chapter, so far, have been at VPY $< 2^{18}$, so $U = 2$.

Any legal X and any Y can be used by the OS.[6] So the XYA architecture permits allocating a rectangular block of data anywhere in virtual memory provided the rectangle spans legal X coordinates. However, grounded rectangles should be sufficient for most 2D software. XYA favors memory allocation close to the ground, which we refer to as **ground affinity**. The semi-direct addressing modes introduced in Section 2.3 are an instance of ground affinity.

2.5 Address translation

The virtual page name is a pair (VPX, VPY), where VPX is a 45-bit chapter name formed by concatenating the 3-bit book number with the 42-bit chapter number, and VPY is a 59-bit *altitude* formed by concatenating $7 - B$ null bits with the Y bits that are not used in the PPO:

$$\begin{aligned} \text{VPX} &= B \times 2^{42} + X_{[41+B:B]} \\ \text{VPY} &= Y_{[63:12-B]} \end{aligned}$$

Figure 2 depicts the extraction of the virtual page name from the virtual address. The page table, stored in memory, allows to translate the virtual page name into a physical page number (**PPN**). The physical address is

$$\text{physical address} = \text{PPN} \times 2^{12} + \text{PPO}_{4\text{K}}$$

where $\text{PPO}_{4\text{K}}$ is given by definition (1).

The page table is organized as a radix tree (aka multi-level page table) where each node is a 4 KB physical page containing 512 PPNs [7, 8]. Figure 3 depicts a page table walk. The page table has five X levels, denoted LX(5), $\cdots$, LX(1), indexed with VPX, and up to seven Y levels LY(7), $\cdots$, LY(1) indexed with VPY. The X levels are accessed in a way similar to conventional multi-level page tables: the root node LX(5) is indexed with the 9-bit value $\text{VPX}_{[44:36]}$, the LX(5) entry points to an LX(4) node which is indexed with the 9-bit value $\text{VPX}_{[35:27]}$, the LX(4) entry points to an LX(3) node, and so on. The last X level, LX(1) contains information about the chapter.

[6]For instance, the OS kernel can map the whole physical memory in a single silo in its address space, virtual coordinate Y being mapped to physical address Y [12, 33].

Y levels are skippable to reduce the length of page table walks.[7] An LX(1) entry contains either a pointer to a Y-level node or the final PPN. The LX(1) entry also contains a 3-bit value U whose meaning is the following:

in this chapter, LX(1) is equivalent to LY(U+1)

This implies that all the accesses to this chapter, so far, have been at altitudes VPY such that

$$\text{VPY} < 2^{9\times U} \tag{2}$$

For instance, if all accesses in a given chapter, so far, have been in the grounded page of this chapter (VPY=0), then $U = 0$ and LX(1) is equivalent to LY(1), meaning that the page table walk terminates at LX(1), which provides the PPN. If $U = 1$, all accesses to the chapter, so far, have been at VPY < 512, and LX(1) is equivalent to LY(2): the LX(1) entry points to a node in LY(1) which is indexed with the 9-bit value $\text{VPY}_{[8:0]}$, and the LY(1) entry contains the final PPN.

If, upon reaching LX(1), the hardware page walker finds that condition (2) does not hold, a page fault is triggered. If the OS determines that the access to this VPY is valid, it updates the page table as follows: it computes the value $U' > U$ corresponding to VPY; it allocates a new node $N(l)$ in each level LY(l) for $l \in [U+1, U']$; it sets the entry of index 0 in node $N(l)$, $l \in [U+1, U']$ to point to $N(l-1)$; it sets the LX(1) entry to point to $N(U')$; it sets $U = U'$ in the LX(1) entry; it allocate new nodes in LY($U-1$), $\cdots$, LY(1) and sets the entries in these nodes corresponding to VPY.

Software using only few silos, in particular 1D software, has a small X footprint: the LX(1) entries are likely to be in the MMU cache [7,8], which allows the page walker to skip the X levels. The MMU cache replacement policy should make it more difficult to evict an LX(1) entry with a high U.

2D software using many silos is expected to have its data close to the ground, so most Y levels are skipped. This is another instance of ground affinity. For example, consider 4 GB of data laid out in book 5 of XYA's memory as a $2^{16} \times 2^{16}$-byte grounded rectangle. We need $2^{16}/2^5 = 2048$ LX(1) entries and $2^{11}/2^9 = 4$ LX(2) entries. As VPY does not exceed $2^{16}/2^7 = 512$, we have $U = 1$ in each of the 2048 LX(1) entries. So the page table walk likely starts with an MMU cache hit at LX(2) and then goes through two levels, LX(1) and LY(1).

2.6 Superpages

XYA allows superpages by stopping address translation at an intermediate page table level [7,8]. Address translation can stop at LY(j) with $j > 1$, or at LX(1) with $U > 0$. For instance, we can use 2 MB pages by stopping translation at LY(2) or at LX(1) with $U = 1$. The superpage width is the same as the page width, i.e., it is 2^B in book B. The superpage height depends on where the address translation stops. If translation stops at LY(j) with $j > 1$, the superpage height is 2^{3+9j-B}. If translation stops at LX(1) with $U > 0$, the superpage height is 2^{3+9j-B} with $j = U + 1$. The superpage PPO is the concatenation of the $9(j-1)$ bits $Y_{[2+9j-B:12-B]}$ with the 12 $\text{PPO}_{4\text{K}}$ bits. For instance, the 2MB PPO is

$$\text{PPO}_{2\text{M}} = Y_{[20-B:12-B]} \times 2^{12} + \text{PPO}_{4\text{K}}$$

[7]Skippable page table levels have been proposed for linear virtual memories [26, 27, 41]. To the best of our knowledge, prevalent ISAs do not support skippable levels, except maybe for a brief comment in the RISC-V manual [35]: "We reserved several PTE bits for a possible extension that improves support for sparse address spaces by allowing page-table levels to be skipped, reducing memory usage and TLB refill latency."

3 XYA microarchitectures

A XYA microarchitecture is mostly similar to a conventional 64-bit CPU microarchitecture and differs only in places where a virtual address is used. We focus the discussion on high-performance CPU microarchitectures, i.e., supporting out-of-order execution.

3.1 2D spatial locality

CPU microarchitectures feature some tables that are searched with virtual addresses, such as translation lookaside buffers (TLB) and branch target buffers (BTB). In a conventional CPU, a set-associative table with 2^s sets, searched with a virtual address X, can be indexed with the value $X_{[s-1:0]}$. Such index function maps 2^s contiguous addresses to 2^s distinct sets, i.e., spatial locality reduces conflicts.

XYA CPUs need a function exploiting 2D spatial locality. More precisely, we are looking for a function that maps a rectangle of area 2^s to 2^s distinct sets. We propose the following integer function:

$$\Phi_s(X,Y) = X_{[0:s-1]} \oplus Y_{[s-1:0]} \tag{3}$$

where $\oplus$ represents the bitwise XOR and notation $X_{[0:s-1]}$ represents the bits $X_{[s-1:0]}$ written backwards, that is, $X_{[0:s-1]} = \sum_{k=0}^{s-1} X_{[k]} 2^{s-1-k}$. The following can be shown:[8]

a rectangle of area 2^s that is x-aligned or y-aligned maps to 2^s distinct $\Phi_s(X,Y)$.

A corollary is that Φ_s maps any grounded rectangle of area 2^s to 2^s distinct sets.

TLBs use full address tags, while some tables like the BTB can use partial tags [15]. Full tags identify the address unambiguously, while partial tags are subject to tag aliasing. For a set-associative table indexed with function (3), a full tag can be obtained by concatenating X and $Y_{[63:s]}$. Partial tags are generally used for speculation mechanisms such as branch prediction or cache way prediction. We can obtain a partial tag as $H(\Phi_n(X,Y))$ where $n > s$ and H is a randomizing hash function (typically an XOR-based one).

3.2 Branch prediction unit

The branch prediction unit generates the speculative PC that is used to fetch instructions. It features various tables containing information about branches past behavior, in particular a BTB, a conditional branch predictor, an indirect jump predictor, and a return address stack.

BTBs generally exploit branch target locality. That is, the branch address and branch target often have several leftmost bits in common [5, 9]. This allows to compress the target address by storing in the BTB entry the rightmost target bits instead of the full target.[9] For a XYA BTB, we can either store in each BTB entry a compressed target PCX and a compressed target PCY, or have a dedicated BTB for oblique jumps.

Modern CPUs generally use an overriding branch prediction scheme [21], with a TAGE-like overriding conditional branch predictor [39]. A TAGE in a XYA CPU needs its index and tag functions to use bits from both PCX and PXY.

[8] Consider a y-aligned rectangle of width 2^a and height 2^{s-a} and two addresses (X,Y) and (X',Y') in this rectangle such that $\Phi_s(X,Y) = \Phi_s(X',Y')$. The rectangle being y-aligned implies that $Y_{[63:s-a]} = Y'_{[63:s-a]}$. Therefore, the a rightmost bits of X and X' are equal, which implies that $|X' - X| < 2^a$ is a multiple of 2^a, so $X' = X$ and $Y' = Y$. A similar reasoning can be used for the case where the rectangle is x-aligned.

[9] For instance, Asheim et al. found that 25 target bits are sufficient for most branches [6].

Modern CPUs feature indirect jump predictors (IJP) such as ITTAGE [38, 39]. The IJP is used for indirect jumps that are not function returns and that are often mispredicted by the BTB. The IJP can use a target compression method similar to the one used in the BTB.

3.3 TLBs

TLB tags are full tags, so a XYA TLB is expected to have larger tags than a conventional TLB. Nevertheless, we can exploit ground affinity to reduce the energy and delay overhead of large tags. The TLB can be split into a small *high* TLB and a larger *low* TLB. The low and high TLBs store translations for pages that have respectively a low and high altitude. The leftmost VPY bits are not stored in the low-TLB tags, they are implicitly equal to zero.

For example, consider a set-associative level-2 low TLB with 128 sets, indexed with $\Phi_7(\mathrm{VPX}, \mathrm{VPY})$ and tagged with $(\mathrm{VPX}, \mathrm{VPY}_{[17:7]})$. The low TLB is used when the 41 bits $\mathrm{VPY}_{[58:18]}$ are all zero. The 11 VPY tag bits allow to put in the low TLB pages of altitude under 2^{18}. In book 0, this corresponds to addresses (X, Y) with $Y < 2^{30}$: the grounded gigabyte in every silo of book 0 uses the low TLB only.

3.4 Load-store unit

Except for TLBs, the load-store unit in a XYA microarchitecture is conventional. We assume that level-1 caches are virtually indexed and physically tagged (VIPT) [8]. That is, the 6 leftmost bits of $\mathrm{PPO}_{4\mathrm{K}}$ are the cache set index, and the 6 rightmost bits of $\mathrm{PPO}_{4\mathrm{K}}$ are the cache line offset, assuming 64-byte lines.[10] As is the case for conventional CPUs with VIPT caches, internal page fragmentation must be kept low so that all cache sets are well utilized.

Modern CPUs have a load queue and a store queue for speculative load execution and store-to-load forwarding [5]. Mispeculation detection entails associative searches with full physical addresses, to avoid the synonym problem [8, 24, 25]. The wider virtual addresses of XYA do not add to the hardware complexity here.[11]

3.5 Load/store address generation

Each address generation unit (AGU) generates a data virtual address and the corresponding *virtual triple* $(\mathrm{VPX}, \mathrm{VPY}, \mathrm{PPO}_{4\mathrm{K}})$. Computing the virtual address requires adding an immediate offset to each coordinate in the general case. Generating the virtual triple requires some shifts, implemented with 8-input multiplexers (as there are 8 books). These shifts make address generation in XYA more complex than in RISC-V. Some architectures with complex addressing modes (x86, ARM,...) may have different load latencies depending on the addressing mode [5, 20]. For instance, in the AMD Zen4, the load latency is one clock cycle longer when address generation needs a 3-input addition or a scaled index [5].

The book number B can be obtained before the addition is completed, by predicting B directly from the input register RX, as crossing a book boundary is expected to be extremely rare. The address generation circuit may provide a fast path for book 0 and/or certain addressing modes where the addition can be bypassed (null IX and null IY or semi-direct).

Among the register based addressing modes, oblique, vertical and horizontal addressing need AGUs with two register read ports while semi-direct, siloed and 1D addressing need a single

[10] It should be noted that, in book 7, a cache line corresponds to rectangle of width 2 and height 32 bytes (see definition (1)), while the cache line in all the other books corresponds to a 64-byte pile.

[11] Virtual address comparisons may be used for accelerating store-to-load forwarding (SLF). However, SLF is speculative and does not need full virtual address bits [24]. For instance, the AMD Zen4 uses the 12 rightmost bits of the virtual address for SLF [5].

register read. Moreover, oblique addressing requires an AGU with two adders.[12] The number of read ports and adders can be reduced by specializing some AGUs with a single register read port and a single adder. Another possibility is to have single-read, single-adder AGUs only and use two AGUs for the more complex addressing modes.

3.6 Data prefetching

Hardware data prefetchers try to predict which data will be accessed in the near future and fetch these data into the caches before memory instructions access them. A prefetcher may work with virtual addresses ("virtual" prefetcher) or with physical line addresses ("physical" prefetcher).

We use the best-offset prefetcher (BOP) as an example [32]. When a line X is accessed, BOP prefetches line $X + O$ where O is a prefetch offset determined dynamically by a training mechanism, which works as follows. On each cache miss or prefetched hit for a line X, one offset o, from a fixed list of offsets, is evaluated by checking in a recent-requests table (RRT) whether line $X - o$ was recently accessed. If this is the case, the score of offset o is increased. At the end of a learning period, the offset with the highest score becomes the new prefetch offset (or prefetch is disabled if the highest score is too low).

The original BOP is a physical prefetcher. We expect a physical BOP in a XYA CPU to be mostly similar to a conventional BOP. A virtual BOP can be devised from the same principles. Below is the description of a virtual BOP for a XYA CPU called *best-vector* prefetcher (BVP).

We define a virtual line as the rectangle in virtual memory corresponding to a physical line. When an access to a virtual line of address (X, Y) generates a miss request or is a prefetched hit, BVP prefetches the virtual line $(X + V_x, Y + V_y)$ where $V = (V_x, V_y)$ is a prefetch vector.[13] The prefetch vector is taken from a fixed list of vectors $v = (v_x, v_y)$. The RRT contains virtual line addresses. A vector v is evaluated by searching $(X - v_x, Y - v_y)$ in the RRT. BVP keeps track of each pending request, miss or prefetch. When a miss request is completed, its virtual line address is written in the RRT. When a prefetch request is completed, its *triggering* address is written in the RRT, where the triggering address of a prefetched line $(X + V_x, Y + V_y)$ is (X, Y).

Multidimensional arrays are easier to prefetch with BVP than with a virtual BOP. Consider for example a huge square matrix $\mathcal{M}$, laid out in conventional linear memory in row-major order [50], and consider prefetching a submatrix of $\mathcal{M}$ with many rows and few columns. The only way for a virtual BOP to prefetch across rows of the submatrix would be to use a large prefetch offset. However, the offset list can only contain a limited number of predefined offsets, and small offsets are preferred because they are more likely to be useful. Now consider matrix $\mathcal{M}$ in XYA's virtual memory, each row being in a distinct silo. The submatrix occupies a long, flat rectangle in virtual memory. A prefetch vector $V = (V_x, 0)$ with a small V_x gives nearly 100% prefetch coverage.

4 The XYC programming language

High-level programming languages (Python, JavaScript,...) will typically hide the dimensionality of the address space from the programmer. Low-level languages such as C, on the other hand, exist for programmers who need high performance and/or precise control over hardware resources.

This section briefly describes a programming language called XYC for writing 2D software. XYC is largely similar to C [22]. Any valid C program is a valid XYC program producing the

[12] The X coordinate in siloed addressing is computed at decode.

[13] V_y is a multiple of 64 in all books but book 7. In book 7, V_y is a multiple of 32 and V_x a multiple of 2.

same result, provided the result does not depend on the size of a pointer. However, a C source compiled with a XYC compiler produces 2D software.

A data type T has width *xsizeof(T)* and height *ysizeof(T)*. An object of type T is mapped in memory as a rectangle whose width and height are that of T. Arithmetic types (char, int, float,...), pointers, structures (*struct*) and unions have unit width and are mapped in memory as piles. **Only arrays can have a width greater than one.**

4.1 Arrays

There are two sorts of arrays: *x-arrays* and *y-arrays*. A y-array is laid out along the Y direction:

```
float a[10]; // a y-array of 10 float
assert(xsizeof(a)==1 && ysizeof(a)==40);
```

An x-array is laid out along the X direction:

```
float a(10); // an x-array of 10 float
assert(xsizeof(a)==10 && ysizeof(a)==4);
```

Note that y-arrays use the same syntax as C, while x-arrays use parentheses '()' instead of square brackets '[]'. The subscript operators '[]' and '()' can be used to access the elements of y-arrays and x-arrays respectively.

A multidimensional array is just an array of arrays, as in C. For example, a 3×2 array can be declared in four different ways, each corresponding to a different layout in memory:

```
int a1[3](2); // xsizeof(a1)=2, ysizeof(a1)=12
int a2(3)[2]; // xsizeof(a2)=3, ysizeof(a2)=8
int a3[3][2]; // xsizeof(a3)=1, ysizeof(a3)=24
int a4(3)(2); // xsizeof(a4)=6, ysizeof(a4)=4
```

An *xy-array* is an x-array of y-arrays, a y-array of x-arrays, or an array of xy-arrays. Arrays *a1* and *a2* above are xy-arrays. Permuting '[]' and '()' in an xy-array declaration (e.g., from [3](2) to (2)[3]) gives a different array with the exact same memory layout. Only unit-width y-arrays can be nested inside a struct or a union.

4.2 Pointers

A *full pointer* is a 128-bit value representing the address of an object:

```
int *p; // full pointer to int
assert(xsizeof(p)==1 && ysizeof(p)==16);
```

The compiler must use two integer registers to hold a full pointer, for the x and y values respectively.

A *ground pointer* is a 64-bit value representing the address of a grounded object. Ground pointers are declared with @ instead of $*$:

```
int @p; // ground pointer to int
assert(xsizeof(p)==1 && ysizeof(p)==8);
```

The dereference operator $*$ is the same for full pointers and ground pointers. Conversion from *T@* to *T** is implicit. Explicit conversion from full pointer to ground pointer extracts the x coordinate. Ground pointer arithmetic operates on the x value, full pointer arithmetic operates on the y value.

For example, the following function returns a pointer to the next element on a diagonal of a matrix:

```
float *oblique_increment(float *p)
{
  float @x = (float@)p;
  uint64_t y = (char*)p - (char*)x;
  x++; // add xsizeof(float)
  p = (float*)((char*)x+y);
  p++; // add ysizeof(float)
  return p;
}
```

The subscript operators '[]' and '()' can be applied to full pointers and, by implicit conversion, to ground pointers: '()' operates on the x coordinate, '[]' on the y coordinate. For example, the function above can be written more concisely:

```
float *oblique_increment(float *p)
{
  return &p(1)+1;
}
```

4.3 Grounding rules

An aggregate type is an array, a struct, or a union with a member of aggregate type. An aggregate object, or *aggregate* for short, is an object of aggregate type. An *outer aggregate* is an aggregate that is not inside another aggregate. XYC guarantees that the following objects are grounded:

- outer aggregates;
- every element of a grounded x-array;
- the first element of a grounded y-array;
- the first member of a grounded struct;
- any member of a grounded union.

The address operator *&* returns a ground pointer to its operand when it is known at compile time that the operand is grounded, otherwise it returns a full pointer. When necessary, a grounded array decays (i.e., is implicitly converted) to a ground pointer to its first element. An array that is not grounded decays to a full pointer to its first element.

The rationale is that the programmer should avoid using full pointers when possible and use ground pointers instead. Indeed, full pointers are twice larger than ground pointers. Moreover, ground pointers simplify alias analysis: if p and q are two ground pointers, *p[1]* and *q[0]* are necessarily distinct objects, whether or not p and q are aliased. For example, consider the following function:

```
void scalar_mul(float @out, float @in, int n, float f)
{
  for (int i=0; i<n; i++)
    out[i] = in[i] * f;
}
```

The compiler can safely vectorize the loop as there are no loop-carried dependencies.[14]

[14]In C, or in XYC with full pointers, the compiler must resort to loop versioning, which introduces overhead [4, 10, 11]. Another solution is for the programmer to create two distinct functions, one using restrict-qualified pointers [49], and a second function for when the output overwrites the input.

5 Memory allocation

5.1 Local variables

Local variables are allocated in the runtime stack. The stack pointer SP holds the X coordinate of a silo. In a function, local variables are mapped in the function's frame, which consists of the silos $X \in [\mathrm{SP}, \mathrm{SP} + \mathrm{F} - 1]$ where F is the frame width of this particular function. The frame is also used to save certain registers, in particular the return PC and registers needed to store temporaries. The function's inputs and output may also be allocated in the frame. SP is decremented by F just after the call and incremented by F just before the return. As frame data are generally few and small, book 7 is the natural place for the stack. When possible, frame data is kept in ground pages.[15] This allows the compiler to use semi-direct addressing.

5.2 Static variables

Static variables may be allocated in different books depending on their type. The book selection for static aggregates depends on the aggregate's width and height. To minimize internal page fragmentation, tall unit-width y-arrays should be allocated in book 0, while wide x-arrays should be allocated in book 7. The compiler uses a default placement that the programmer can override with a variable attribute (e.g., *__attribute__ ((book (7)))*). We assume the following default placement, which we call *square-of-pages*:

> If the aggregate's area is less than the page size (4 KB), the selected book is the highest book number such that the page height is greater than or equal to the aggregate's height. Otherwise, if the aggregate's area exceeds the page size, the selected book is the book where the page aspect ratio is the closest to the aggregate's aspect ratio.

Static scalar variables can be mapped in any book, preferably in ground pages already populated. The compiler can take advantage of semi-direct addressing (Section 2.3). Static scalars can also be mapped close to the code (most likely in book 0), where PC-relative addressing can be used.

5.3 Dynamic allocation

The XYC standard library must provide functions for dynamic allocation, for example, mimicking C:

```
void @ xy_malloc(size_t width, size_t height, int book);
void xy_free(void @ptr);
```

Notice that *xy_malloc* returns a ground pointer, i.e., the allocated memory region is a set of adjacent silos. This is consistent with XYC's grounding rules (Section 4.3).

If a valid book number is passed as last argument of *xy_malloc*, the returned pointer points to a silo in the specified book. Otherwise, if an invalid book number is passed, the book is selected according to the square-of-pages placement (Section 5.2). The height parameter is used by the square-of-pages placement. The height is also used by the allocator to estimate the number of physical pages that have been mapped in each chapter, so as to do a better job of reusing freed silos.

[15] This is always possible for objects whose height does not exceed 32 bytes (the page height in book 7), in particular scalar variables.

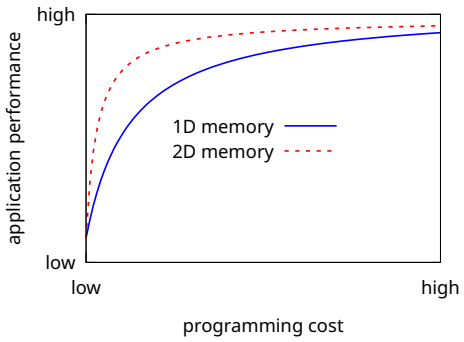

Figure 4: Application performance depends on programming cost (fake curves).

In C, the *realloc* function can be used to grow or reduce a previously allocated memory block, e.g., to implement a dynamic array [47]. If the memory addresses after the block are free, the block may be grown in place, i.e., without needing relocation. Otherwise, the block is moved to a new address, either by doing a memory copy or by a call to *mremap* [17].

On a XYA system, as *xy_malloc* allocates full silos, a dynamically allocated object can always grow in place along the Y direction. Nevertheless, the following function should be called to grow an object:

```
void @ xy_realloc(void @ptr, size_t height);
```

This function keeps the allocator up to date with the maximum altitude in each chapter. When *ptr* and *height* are not null, *xy_realloc* always returns the input *ptr*. That is, the address of the object stays the same throughout its lifetime.[16]

6 The programmer's point of view

While application performance is an important objective in many situations, it is never the sole factor determining how a program is written. Other factors such as development time, programmer's skill, portability, maintainability, ease of use (for a library), security, etc., are important too. When performance is a primary objective, these other factors can be viewed collectively as the programming cost. Programming cost is not easily quantifiable, yet it is tangible to software developers and their employers. If we could quantify the programming cost, we could plot performance-vs-cost curves, as illustrated in Figure 4 (with fake curves). Most applications with performance as primary objective lie probably somewhere in the knee of the curve. Only a few applications, such as certain libraries, push performance in the plateau past the knee. Yet, even performance-focused libraries may stay in the knee. As an example, there can be significant performance differences between popular BLAS-like libraries [54]. For instance, the developers of the BLIS library considered portability among their primary objectives, besides raw performance [43].

Table 2 lists six features of XYA that programmers can exploit to improve performance at equal programming cost or reduce programming cost at equal performance. The first four features have been mentioned earlier, elaborating on them is left for future work. This section focuses on the last two features: xy-arrays and 2D pages.

[16] With a linear memory, guaranteed in-place reallocation is possible but requires reserving a pessimistic amount of virtual memory for each object, which leads to internal page fragmentation.

feature	description	benefit
128-bit address	huge address space (Section 2.4)	security
siloed addressing	for code sandboxing (sections 2.2 2.3)	security
ground pointer	alias analysis facilitated (Section 4.3)	performance
in-place realloc	grow object without relocation (Section 5.3)	performance
xy-array	natural memory layout for 2D objects	productivity
2D page	multiple page aspect ratios	performance

Table 2: XYA features.

```
int *argmax(int *val, int n)
{
  int imax = 0;
  for (int i=1; i<n; i++)
    if (val(i) > val(imax))
      imax = i;
  return &val(imax);
}

struct person @oldest(struct person @population, int size)
{
  return (void@)argmax(&population->age, size);
}
```

Figure 5: An x-array of structs can be viewed as a struct of x-arrays.

6.1 Multidimensional xy-arrays

The simplest and most commonly used layouts for a 2D object (a dense matrix, an image, ...) in a 1D memory are row-major and column-major [50]. For instance, with the row-major layout, each row occupies a contiguous block of memory, and elements in the same column are at addresses separated by a column stride equal to the row size. If we want to define a C function operating on any submatrix, the function must take as parameter the column stride of the matrix. A 2D address space allows to represent a matrix as a rectangle in memory, and there is no distinction between a matrix and a submatrix, which simplifies the implementation of blocked algorithms (e.g., high-performance matrix multiplication).

A 2D address space provides a natural memory mapping not only to 2D arrays but also 1D arrays of structs, which are 2D data structures in disguise. More specifically, in XYC, an x-array of structs can be viewed as a struct of x-arrays. Figure 5 shows an example XYC function finding the oldest person in a population by calling a simple *argmax* function taking an x-array of integers as argument. This can be done in C but in a less elegant and more error-prone way.[17] Somehow, a 2D address space solves the array-of-struct vs. struct-of-array dilemma [46].

A 2D address space offers to the programmer many more options for selecting the best memory layout for representing a multidimensional object. For example, consider a three-dimensional (3D) grid of length L, width W and height H, represented as a 3D array. In C, the programmer can choose among $3! = 6$ possible array layouts, which is the number of permutations of L, W, H. A performance aware programmer will select the most appropriate layout depending on L,

[17] In C, we must pass the size of the struct to *argmax* and subtract the struct member offset from the returned pointer.

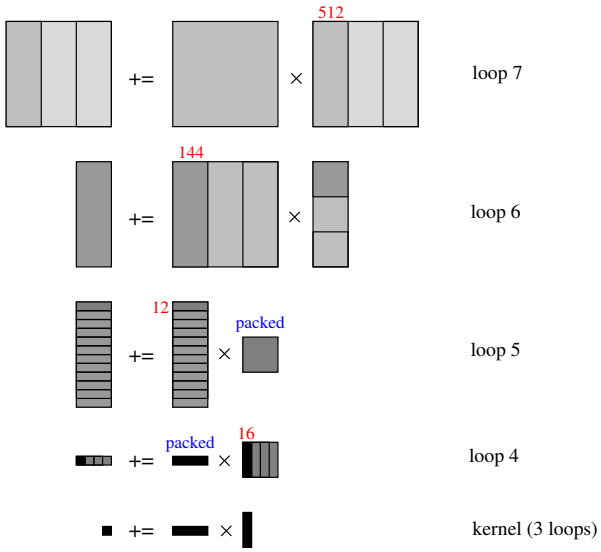

Figure 6: DGEMM-lite: single-threaded matrix multiplication specialized for an Intel Cascade Lake CPU (the picture is not to scale).

W, H and how the grid is used. In XYC, there are 4 different combinations of x-arrays and y-arrays for a 3D array, ()()(), ()()[], ()[][], or [][][], and there are 6 permutations of L, W, H per combination. So there are $4 \times 3! = 24$ different layouts that the programmer can choose from.

6.2 2D pages

The programmer can select the book, i.e., the page aspect ratio, for each static or dynamically allocated array. This gives the programmer as many degrees of freedom for performance optimization as the number of such arrays in the program. As an example, this section considers high-performance matrix multiplication.

We consider the matrix multiplication C = A × B where A and B are $N \times N$ square matrices of double-precision floating-point values. Assuming the standard matrix multiplication method, this represents N^3 scalar fused multiply-add (FMA) operations. That is, each element of A and each element of B is read N times. For large N, this means much memory reuse that TLBs and caches can exploit. However, approaching the peak performance, i.e., saturating the FMA units, is not a trivial task.

High-performance multiplication of large matrices on CPUs is generally implemented along the lines of the GotoBLAS library [18, 19, 48]. The C submatrix written by the kernel is kept in vector registers, and the A and B submatrices read by the kernel have been *packed*, i.e., they have been copied in contiguous memory regions [18, 19]. Without packing, the performance of large matrix multiplications is hurt by TLB misses and, possibly, cache misses [18, 36].

We wrote, in C, a simplified matrix multiplication called DGEMM-lite, depicted in Figure 6. DGEMM-lite is single-threaded and specialized for an Intel Cascade Lake CPU. DGEMM-lite works with matrices in row-major format. The kernel does 144 outer products (12-long subcolumn of A times 16-long subrow of B). The dimensions of the kernel submatrices shown in Figure 6 depend on the ISA and microarchitecture. Therefore, in the general case, the matrix dimensions are not multiples of the kernel submatrices dimensions. This leads to edge cases that

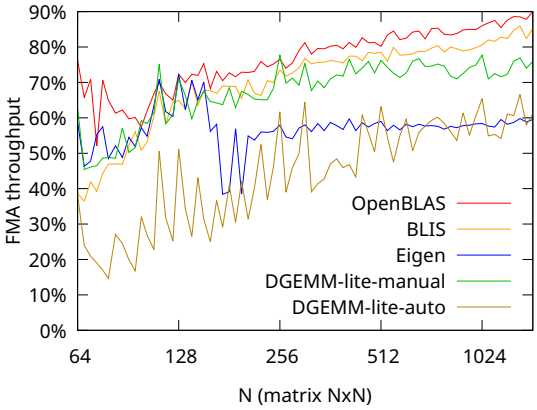

Figure 7: FMA throughput, on a single Intel Cascade Lake core, of double-precision square-matrix multiplications from several libraries, against DGEMM-lite.

are usually dealt with by implementing extra kernels or by zero-padding [54]. DGEMM-lite uses the latter approach. We pack the A and B kernel submatrices in fixed-size buffers. The packing of the B submatrix is done before entering loop 5 while the packing of the A submatrix is done before entering loop 4 (see Figure 6). For the B submatrix, we use the packing method first implemented in GotoBLAS, i.e., the submatrix elements are stored in the buffer in the order they are read by the kernel [18].

We implemented two versions of DGEMM-lite. In the first version, called *DGEMM-lite-auto*, the kernel and packing are written in C and auto-vectorized with GCC 12.2. In the second version, called *DGEMM-lite-manual*, the kernel and packing use AVX-512 intrinsics, manual software prefetch and, for the kernel, manual loop unrolling.

We measured the FMA throughput of DGEMM-lite on a single Intel Cascade Lake core, using $N \times N$ matrices. The Cascade Lake core we use can execute 16 scalar double-precision FMAs per clock cycle at a clock frequency of 3.4 GHz. This defines a shortest time for the execution of N^3 scalar FMAs, which defines 100% FMA throughput. We execute the same matrix multiplication multiple times until the total time exceeds one second. Figure 7 compares the FMA throughput of DGEMM-lite against double-precision matrix multiplication from the OpenBLAS [3, 52], BLIS [1, 43] and Eigen [2] libraries. The performance of DGEMM-lite-manual is slightly less than that of OpenBLAS and BLIS, especially on large matrices.[18] Auto-vectorization is not quite as effective as manual vectorization, although the performance of DGEMM-lite-manual is on a par with that of Eigen for large matrices.

Figure 8 shows the performance of DGEMM-lite-manual on an Intel Cascade Lake core, with and without packing. Packing is almost always beneficial, bringing more than 50% speedup for certain matrix sizes. The packing of B is important for performance. We verified with performance counters that its main effect is to reduce the number of TLB misses. The packing of A has less effect in general. It is most effective when N is large and a power of two (512 and 1024 in Figure 8). We verified with performance counters that its main effect is a reduction of the number of level-1 data cache misses.

Having established that DGEMM-lite has realistic performance, we use it as a benchmark to

[18]The kernels of OpenBLAS and BLIS are written in assembly, which allows better control of instruction scheduling and register allocation than intrinsics. Moreover, the outer loop in the GotoBLAS GEMM partitions A and B simultaneously [18, 43, 52], which allows to amortize the cost of packing A for very large matrices.

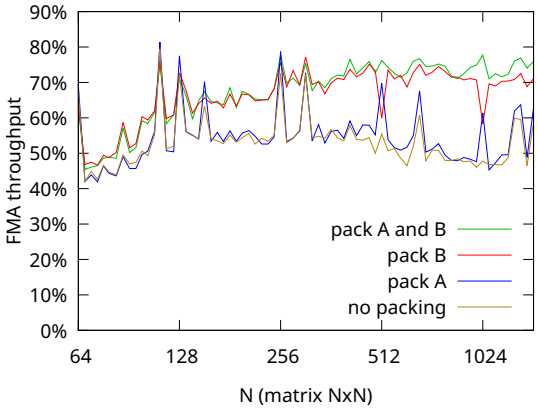

Figure 8: Performance of DGEMM-lite-manual on an Intel Cascade Lake core, with and without packing.

```
pointer<var<int>> argmax(pointer<var<int>> val, int n)
{
  int imax = 0;
  for (int i=1; i<n; i++)
    if (val(i) > val(imax))
      imax = i;
  return &val(imax);
}

xptr<person> oldest(xptr<person> population, int size)
{
  return (xpointer<void>) argmax(&population->age,size);
}
```

Figure 9: Emulating the XYC code of Figure 5 with MXY custom types.

study matrix multiplication on a XYA CPU. In particular, we seek to understand if packing is needed on a XYA CPU.

As we do not have a XYC compiler, we have written a C++ library, called MXY, allowing to emulate programming in XYC. Figure 9 shows how the code of Figure 5 can be emulated with MXY. MXY defines custom types leveraging C++ templates, constructors, destructors, operator overloading and user-defined conversions. Static objects are allocated in memory when they are constructed. Allocation/deallocation in the simulated runtime stack is done at construction/destruction of local objects.

A C++ program using MXY custom types can generate a trace of all data accesses. The data address trace obtained with MXY is not exactly identical to the data addresses that a true XYA executable would access, as a XYC compiler would do optimizations such as register allocation and auto-vectorization. Regarding register allocation, MXY can eliminate reads and writes to some temporary values identified by move constructors and move assignment operators. Register allocations not captured by move constructors/assignments can be simulated manually by the programmer. Each XYC type in MXY has a register-like counterpart. For instance, the C++ type *int* is the register-like counterpart of the XYC type *var<int>*, and MXY can do

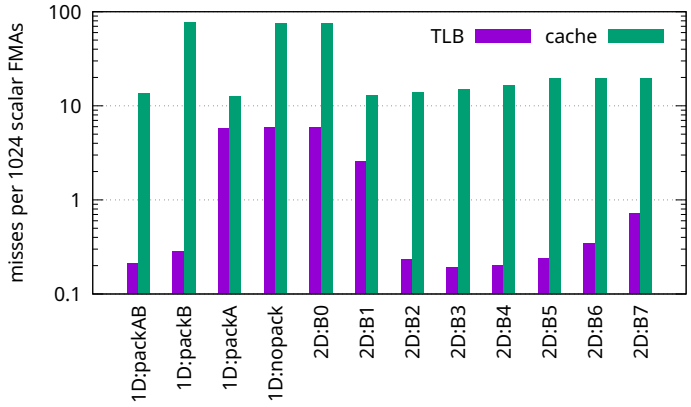

Figure 10: TLB/cache misses per 1024 scalar FMAs when multiplying 1024×1024 matrices: DGEMM-lite-1D (4 leftmost pairs of bars, with and without packing) vs. DGEMM-lite-2D (8 rightmost pairs of bars, book $B \in [0, 7]$). The scale on the y-axis is logarithmic.

implicit conversions from *int* to *var<int>* and conversely: reading or writing a *var<int>* outputs an address, while accessing an *int* does not.

To validate our methodology, we wrote a second library, called MX, similar to MXY but defining custom types corresponding to C and a conventional 1D address space. We obtained a data address trace for DGEMM-lite-auto with a Pintool [28]. Then, using MX, we wrote a pseudo-C version of DGEMM-lite-auto called *DGEMM-lite-1D*. We fed the DGEMM-lite-auto and DGEMM-lite-1D traces to a TLB/cache simulator and verified that the *number* of cache/TLB misses from the two traces match approximately. To gain more confidence in the methodology, we did similar verifications with a few other benchmarks (quicksort, FFT, LU factorization, hash table).

Then, using MXY, we wrote a pseudo-XYC version of DGEMM-lite-auto called *DGEMM-lite-2D*. DGEMM-lite-2D mimics DGEMM-lite-1D but with two key differences:

- DGEMM-lite-2D uses xy-arrays for the matrices instead of the row-major format: an $N \times N$ matrix occupies N consecutive silos, each silo holding one row of the matrix;
- DGEMM-lite-2D does not implement packing: the kernel reads the A and B submatrices directly.

We wrote a modified version of the TLB/cache simulator to simulate the TLB and data cache of a XYA microarchitecture, as described in Section 3. Then we ran TLB/cache simulations to compare DGEMM-lite-1D and DGEMM-lite-2D. The TLB and cache are modeled after the Intel Cascade Lake: the TLB is 64-entry, 4-way set-associative (i.e., 16 TLB sets) and the data cache is 32 KB, 8-way set-associative (i.e., 64 cache sets). The cache line is 64 B, the page 4 KB.

Figure 10 shows the number of TLB/cache misses per 1024 scalar FMAs when multiplying 1024×1024 matrices.[19] The leftmost 4 pairs of bars correspond to DGEMM-lite-1D with and without packing. The previous performance analysis with the Cascade Lake core is confirmed: the packing of A reduces the number of cache misses and the packing of B reduces the number of TLB misses.

The 8 rightmost pairs of bars correspond to DGEMM-lite-2D, varying the book B where all three matrices (A,B,C) are placed. It can be observed that book 0 is equivalent to a conventional

[19] 1024 scalar FMAs corresponds to 64 clock cycles on a Cascade Lake core when FMA throughput is 100%.

1D address space, in terms of TLB/cache misses. Placing the matrices in books $B \in [2, 5]$ is approximately equivalent to packing B, in terms of TLB misses. The use of 2D pages also reduces the number of cache conflict misses. The cache is indexed with the 6 leftmost bits of PPO_{4K} (Figure 2), so selecting a book is like selecting a cache index function. The A submatrix consists of 12 subrows: in book 0, 12 elements in the same column have the same Y coordinate and are therefore mapped in the same cache set. As the cache associativity is less than 12, conflict misses occur. When A is in book $B > 0$, some X bits are used in the indexing, which implies that 12 consecutive elements in a column are mapped to multiple cache sets, which removes same-column conflicts. The book selected by the default square-of-pages placement is book 5 here, which is close to optimal. However, manually selecting book 2 or 3 should yield slightly better performance.

The fundamental reason why packing is not needed for matrix multiplication on a XYA CPU is that submatrices of A and B are rectangles in the 2D address space, and the TLB and level-1 data cache of a XYA CPU are designed so that rectangles fit in.

Programming a high-performance multiplication for a XYA CPU is simpler, for two reasons. First, we do not need to implement packing. Second, while the cost of packing can be made negligible for large squarish matrices, it can be significant when matrices are small or not squarish [18, 54]. Taking care of this problem requires more effort from the programmer [53].

7 Related work

Segmentation was introduced in the 1960's as a way to enlarge the virtual address space [13,14,34]. ISAs using segmentation have segment registers. A virtual address is formed by combining the content of a segment register with the result of a linear address computation. Segmentation defines a collection of linear address spaces, it was not intended to provide a 2D address space to programmers. ISAs using segmentation do not allow ALU operations to be performed directly on segment registers and there is no notion of spatial locality across segments.

8 Conclusion

We have described a hypothetical architecture called XYA whose main novelty is a two-dimensional address space partitioned into regions called books, each book corresponding to a distinct 2D page aspect ratio. We have described the main differences between the microarchitecture of a XYA CPU and that of a conventional CPU. We have proposed a C-like programming language, called XYC, allowing programmers to take advantage of the 2D address space. We have identified several features of XYA that programmers can exploit to improve performance at equal programming cost or reduce programming cost at equal performance.

For the moment, XYA exists only as an idea, and an incomplete one. We have briefly mentioned a few implications of XYA for the compiler and the operating system. We have only scratched the surface of the problem, the implications certainly go much deeper.

Before investing resources to develop a XYA simulator and a compiler, another problem deserves serious consideration, that of porting existing software to XYA. The porting of 32-bit software to 64-bit architectures was not straightforward [31]. We have no reason to believe that the porting of existing software to XYA would be easier.

References

[1] "BLIS." [Online]. Available: https://github.com/flame/blis

[2] "Eigen." [Online]. Available: http://eigen.tuxfamily.org/

[3] "OpenBLAS." [Online]. Available: https://www.openblas.net/

[4] P. Alves, F. Gruber, J. Doerfert, A. Lamprineas, T. Grosser, F. Rastello, and F. M. Q. Pereira, "Runtime pointer disambiguation," in *International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2015.

[5] AMD, "Software optimization guide for the AMD Zen4 microarchitecture," Jan. 2023, publication 57647. [Online]. Available: https://www.amd.com

[6] T. Asheim, B. Grot, and R. Kumar, "A storage-effective BTB organization for servers," in *International Symposium on High-Performance Computer Architecture (HPCA)*, 2023.

[7] T. W. Barr, A. L. Cox, and S. Rixner, "Translation caching: skip, don't walk (the page table)," in *International Symposium on Computer Architecture (ISCA)*, 2010.

[8] A. Bhattacharjee and D. Lustig, *Architectural and Operating System Support for Virtual Memory*. Morgan & Claypool, 2018.

[9] B. Calder and D. Grunwald, "Fast and accurate instruction fetch and branch prediction," in *International Symposium on Computer Architecture (ISCA)*, 1994.

[10] K. Chitre, P. Kedia, and R. Purandare, "The road not taken: exploring alias analysis based optimizations missed by the compiler," in *Proceedings of the ACM on Programming Languages, Volume 6, OOPSLA2*, Oct. 2022.

[11] K. Chitre, P. Kedia, and R. Purandare, "RAPID: region-based pointer disambiguation," in *Proceedings of the ACM on Programming Languages, Volume 7, OOPSLA2*, Oct. 2023.

[12] J. Corbet, "Reconsidering the direct-map fragmentation problem," may 2023. [Online]. Available: https://lwn.net/Articles/931406/

[13] R. C. Daley and J. B. Dennis, "Virtual memory, processes, and sharing in MULTICS," *Communications of the ACM*, vol. 11, no. 5, May 1968.

[14] J. B. Dennis, "Segmentation and the design of multiprogrammed computer systems," *Journal of the ACM*, vol. 12, no. 4, Oct. 1965.

[15] B. Fagin and K. Russell, "Partial resolution in branch target buffers," in *International Symposium on Microarchitecture (MICRO)*, 1995.

[16] B. Ford and R. Cox, "Vx32: lightweight user-level sandboxing on the x86," in *USENIX Annual Technical Conference*, 2008.

[17] GNU C library, "Malloc." [Online]. Available: https://sourceware.org/glibc/wiki/MallocInternals

[18] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software*, vol. 34, no. 3, May 2008.

[19] K. Goto and R. A. van de Geijn, "High-performance implementation of the level-3 BLAS," *ACM Transactions on Mathematical Software*, vol. 35, no. 1, Jul. 2008.

[20] Intel, "Optimizing earlier generations of Intel 64 and IA-32 processor architectures, throughput, and latency," Apr. 2024, document 356477-050US. [Online]. Available: https://www.intel.com

[21] D. A. Jiménez, S. W. Keckler, and C. Lin, "The impact of delay on the design of branch predictors," in *International Symposium on Microarchitecture (MICRO)*, 2000.

[22] B. W. Kernighan and D. M. Ritchie, *The C programming language*, 2nd ed. Prentice Hall, 1988.

[23] T. Kilburn, D. Edwards, M. Lanigan, and F. Sumner, "One level storage system," *IRE Transactions on Electronic Computers*, vol. EC-11, no. 2, Apr. 1962.

[24] J. M. King and M. S. Sobel, "Techniques for performing store-to-load forwarding," Patent US11113056B2, Sep. 2021, filed nov. 27, 2019.

[25] D. Leibholz and R. Razdan, "The Alpha 21264: a 500 MHz out-of-order execution microprocessor," in *IEEE COMPCON*, 1997.

[26] J. Liedtke, "Address space sparsity and fine granularity," *ACM SIGOPS Operating System Review*, vol. 29, no. 1, Jan. 1995.

[27] J. Liedtke and K. Elphinstone, "Guarded page tables on Mips R4600 or an exercise in architecture-dependent micro optimization," *ACM SIGOPS Operating System Review*, vol. 30, no. 1, Jan. 1996.

[28] 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 *Conference on Programming Language Design and Implementation (PLDI)*, 2005.

[29] V. B. Lvin, G. Novark, E. D. Berger, and B. Zorn, "Archipelago: trading address space for reliability and security," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[30] H. Marco-Gisbert and I. Ripoll-Ripoll, "Address space layout randomization next generation," *Applied Sciences*, vol. 9, no. 14, Jul. 2019.

[31] J. R. Mashey, "The long road to 64 bits," *ACM Queue*, vol. 4, no. 8, oct 2006.

[32] P. Michaud, "Best-offset hardware prefetching," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[33] A. Panwar, A. Prasad, and K. Gopinath, "Making huge pages actually useful," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[34] B. Randell and C. J. Kuehner, "Dynamic storage allocation systems," *Communications of the ACM*, vol. 11, no. 5, May 1968.

[35] RISC-V International, "The RISC-V instruction set manual," vol. II: Privileged architecture, version 20240411. [Online]. Available: https://riscv.org/technical/specifications/

[36] C. S. Rohwedder, N. Henderson, J. P. L. de Carvalho, Y. Chen, and J. N. Amaral, "To pack or not to pack: a generalized packing analysis and transformation," in *International Symposium on Code Generation and Optimization (CGO)*, 2023.

[37] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary CPU architectures," in *USENIX Security Symposium*, 2010.

[38] A. Seznec, "A 64-Kbytes ITTAGE indirect branch predictor," in *JILP Workshop on Computer Architecture Competitions (JWAC-2): Championship Branch Prediction*, 2011. [Online]. Available: https://jilp.org/jwac-2/

[39] A. Seznec and P. Michaud, "A case for (partially) tagged geometric history length branch prediction," *The Journal of Intruction-Level Parallelism*, vol. 8, Feb. 2006.

[40] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *ACM Conference on Computer and Communication Security (CCS)*, 2004.

[41] C. Szmajda and G. Heiser, "Variable radix page table: a page table for modern architectures," in *Advances in computer systems architecture, LNCS*, vol. 2823. Springer, 2003.

[42] A. S. Tanenbaum and T. Austin, *Structured computer organization*, 6th ed. Pearson, 2013.

[43] F. G. Van Zee, T. M. Smith, B. Marker, T. M. Low, R. A. van de Geijn, F. D. Igual, M. Smelyanskiy, X. Zhang, M. Kistler, V. Austel, J. A. Gunnels, and L. Killough, "The BLIS framework: experiment in portability," *ACM Transactions on Mathematical Software*, vol. 42, no. 2, Jun. 2016.

[44] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Symposium on Operating Systems Principles (SOSP)*, 1993.

[45] D. Weaver and S. McIntosh-Smith, "An empirical comparison of the RISC-V and AArch64 instruction sets," in *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC-W)*, 2023.

[46] Wikipedia, "AoS and SoA," Jul. 2024. [Online]. Available: https://en.wikipedia.org/wiki/AoS_and_SoA

[47] Wikipedia, "Dynamic array," Jul. 2024. [Online]. Available: https://en.wikipedia.org/wiki/Dynamic_array

[48] Wikipedia, "GotoBLAS," Jul. 2024. [Online]. Available: https://en.wikipedia.org/wiki/GotoBLAS

[49] Wikipedia, "Restrict," Jul. 2024. [Online]. Available: https://en.wikipedia.org/wiki/Restrict

[50] Wikipedia, "Row- and column-major order," Jul. 2024. [Online]. Available: https://en.wikipedia.org/wiki/Row-_and_column-major_order

[51] Wikipedia, "Unary coding," Jul. 2024. [Online]. Available: https://en.wikipedia.org/wiki/Unary_coding

[52] Z. Xianyi, W. Qian, and Z. Yunquan, "Model-driven level 3 BLAS performance optimization on Loongson 3A processor," in *International Conference on Parallel and Distributed Systems*, 2012.

[53] R. G. Xu, F. G. Van Zee, and R. A. van de Geijn, "Towards a unified implementation of GEMM in BLIS," in *International Conference on Supercomputing (ICS)*, 2023.

[54] W. Yang, J. Fang, and D. Dong, "Characterizing small-scale matrix multiplications on ARMv8-based many-core architectures," in *International Parallel and Distributed Processing Symposium (IPDPS)*, 2021.

[55] C. Yarvin, R. Bukowski, and T. Anderson, "Anonymous RPC: low-latency protection in a 64-bit address space," in *USENIX Summer Technical Conference*, 1993.

[56] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "Native Client: a sandbox for portable, untrusted x86 native code," *Communications of the ACM*, vol. 53, no. 1, Jan. 2010.

RESEARCH CENTRE
Centre Inria de l'Université de Rennes

Campus universitaire de Beaulieu
Avenue du Général Leclerc
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399