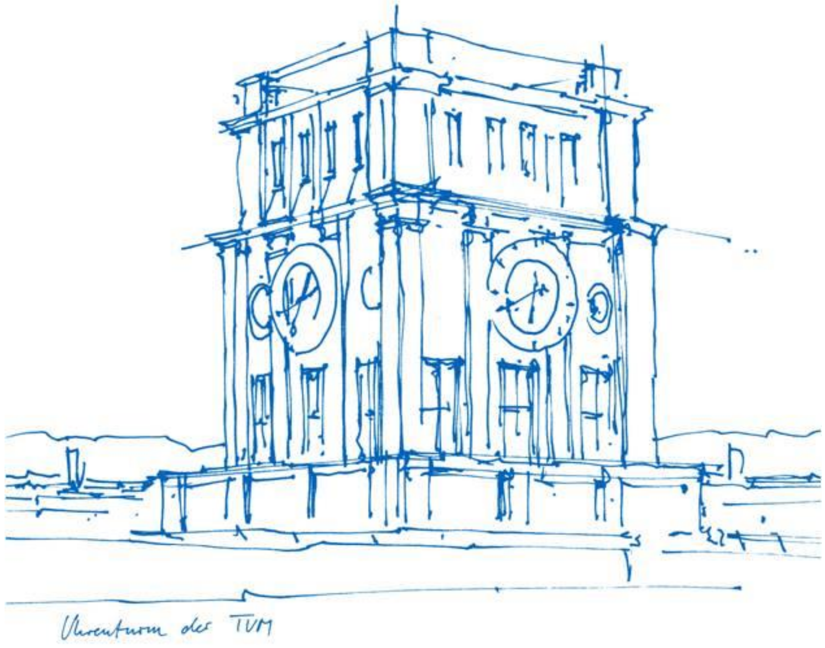
Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Technische Universität München - Lehrstuhl für Rechnerarchitektur und parallele Systeme

Peano-Kurven (A214)

Mohamed Attia, Thomas Torggler, Patrick Zimmermann

München, der 16. Februar 2021

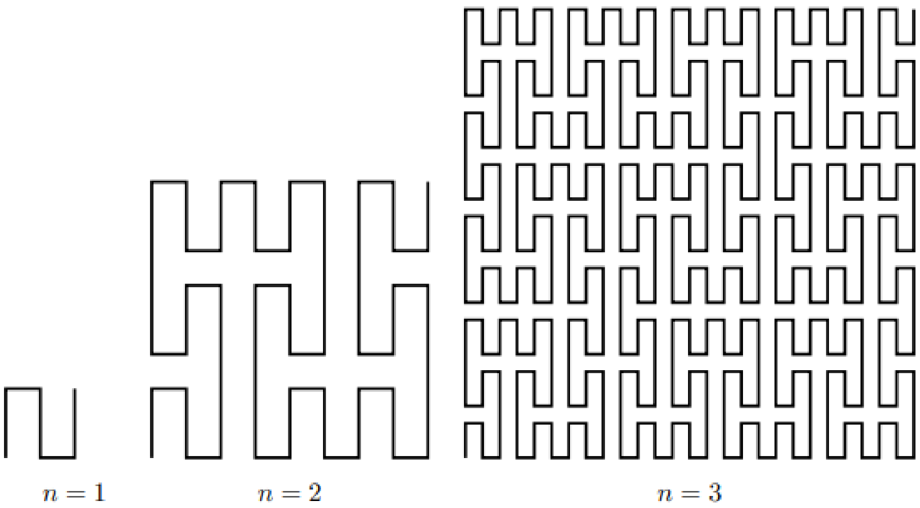


Motivation: Beispiel des Ansatzes

-

Motivation: Raumfüllende Kurven

-

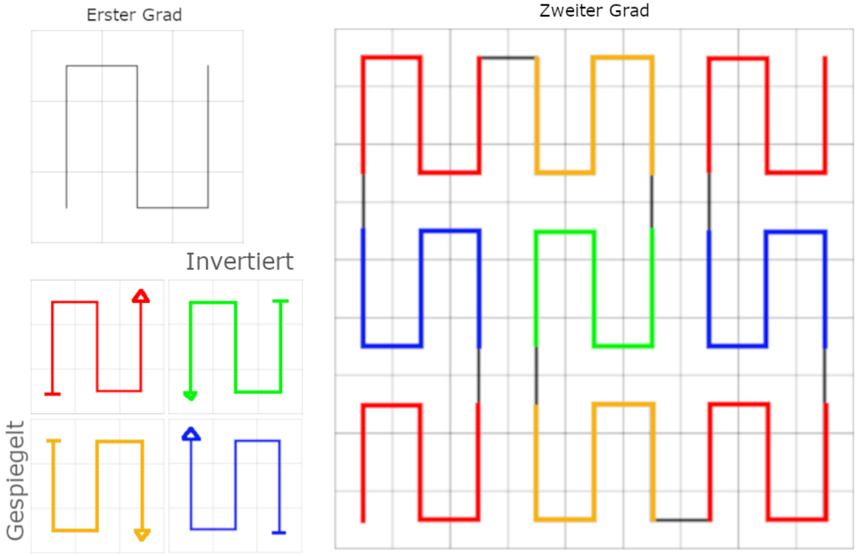


Lösungsansatz

```
int calcNextInplace(int currGrad, int []curr, int pos)
     size   ->  9^currGrad
    curr[pos]   ->  0   // First Step between Permutations upwards
    reverseMirrorInPlace(curr, pos, size)   // second step
    pos  -> pos + size
    curr[pos]   ->  0     //Step between Permutations upwards
    copyInPlace(curr, pos, size)     // third step
     pos  -> pos + size
    curr[pos]   ->  1     //Step between Permutations rigth
    mirrorInPlace(curr, pos, size)     //4th Step
     pos  -> pos + size
    curr[pos]   ->  2     //Step between Permutations downwards
    reverseInPlace(curr, pos, size)     //5th Step
     pos  -> pos + size
    curr[pos]   ->  2     //Step between Permutations downwards
    mirrorInPlace(curr, pos, size)     //6th Step
     pos  -> pos + size
    curr[pos]   ->  1     //Step between Permutations rigth
    copyInPlace(curr, pos, size)     //7th Step
     pos  -> pos + size
    curr[pos]   ->  0     //Step between Permutations upwards
    reverseMirrorInPlace(curr, pos, size)     //8th Step
     pos  -> pos + size
    curr[pos]   ->  0     //Step between Permutations upwards
    copyInPlace(curr, pos, size)     //last Step
     pos  -> pos + size
    return pos
ENDFUNCTION
```

```
void peanoInPlace(int grad, int[] x1, int[] y1)
     if (grad <= 0)
      printf("Error number not valid !")
      return
    ENDIF
    size   ->  9^grad
    int []array   -> new int [size]      // Allocate Directionarray and hardcode first curve
    if (array == NULL)
      perror("Please try again with a smaller degree ")
    ENDIF
    array[1..8]  ->  { 0, 0, 1, 2, 2, 1, 0, 0}
     pos   ->  9
     currGrad   ->  1
    do
      pos   ->  calcNextInplace(currGrad, array, pos)
      currGrad++
    while(currGrad < grad)
     x   ->  1
     y   ->  1
    x1[0]   ->  x
    y1[0]   ->  y
    for ( i ->  1 ; i < size ; i++)
      switch (array[i])
      case 0: //up
          y++
          break
      case 2: //down
          y--
          break
      case 3: //left
          x--
          break
      case 1: //right
          x++
          break
      default:
          break
      x1[i]   ->  x
      y1[i]   ->  y
    ENDFOR
ENDPROCEDURE
```

Lösungsansatz: Richtungsarray



- Data centric processing
- Data pushed towards operator
- Queries compiled into native machine code (LLVM)
- Maximize data and code locality

Korrektheit:

-

Performanzanalyse:

-

Performanzanalyse:

-

Fazit: Verbesserungsmöglichkeiten

-

Advanced Parallelization Techniques: 1st Type

1) Inter-tuple parallelism: SIMD registers

✓ SIMD instructions => speed up processing

✓ Delay branching

- BUT …

- LLVM directly allows for modelling SIMD values as vector types

⇒ The impact is relatively minor

Evaluation

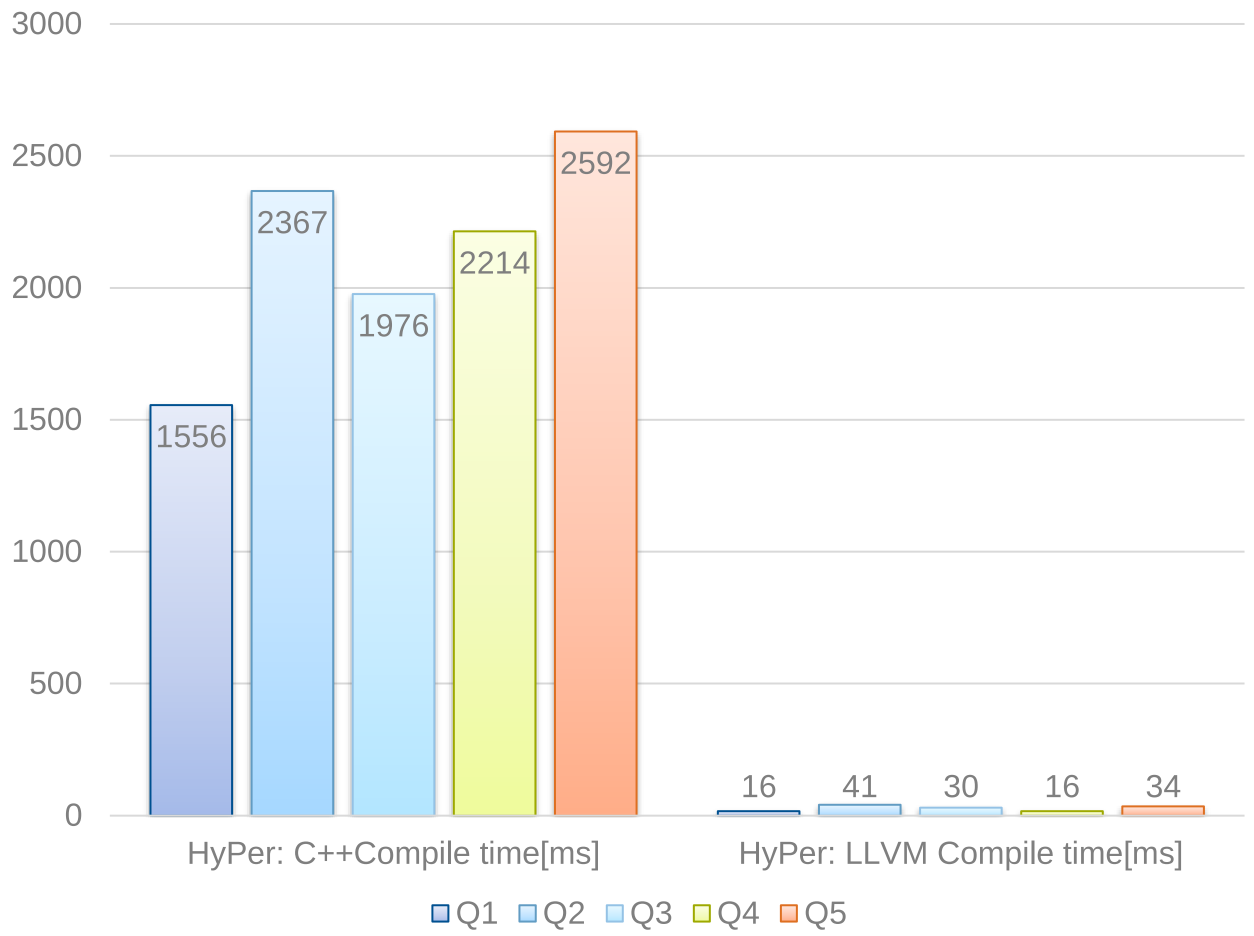
➢ We implemented this in our HyPer system

- initially we generated C++ code from code fragments
- then, switched to the data-centric LLVM code: comparison C++ vs. LLVM

Evaluation: C++ <==> LLVM

	Q1	Q2	Q3	Q4	Q5
HyPer: C++ [ms]	142	374	141	203	1416
Compile time[ms]	1556	2367	1976	2214	2592
HyPer: LLVM [ms]	35	125	80	117	1105
Compile time[ms]	16	41	30	16	34
VectorWise [ms]	98	-	257	436	1107
MonetDB [ms]	72	218	112	8168	12028
DB X [ms]	4221	6555	6410	3830	15212

Evaluation: C++ <==> LLVM



Evaluation

➢ We implemented this in our HyPer system

- initially we generated C++ code from code fragments
- then, switched to the data-centric LLVM code: comparison C++ vs. LLVM

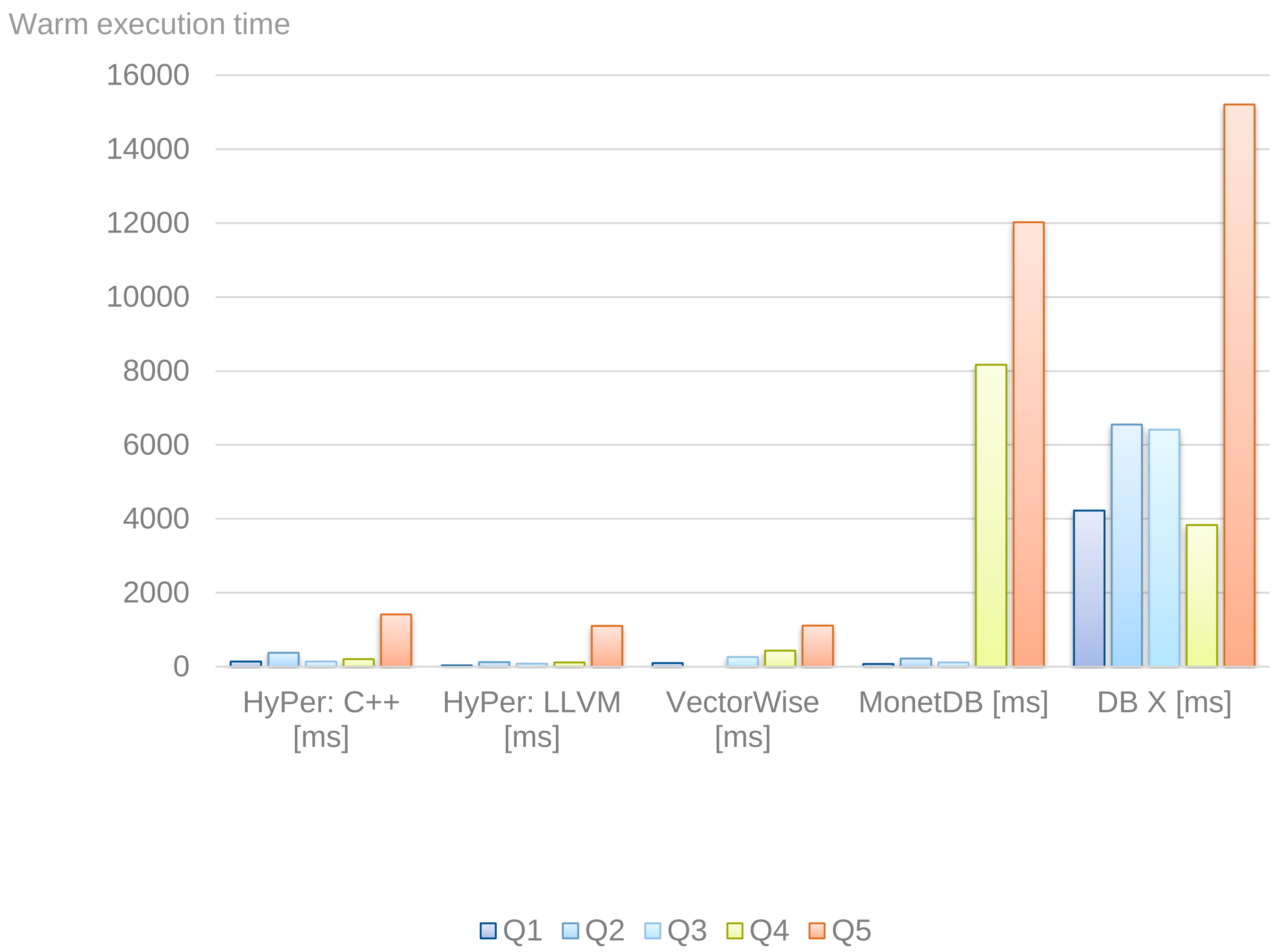
➢ Compared it with other systems

- MonetDB 1.36.5, Ingres VectorWise 1.0, DB X (commercial DBS)
- 5 TPC-H queries (Q1,2,3,4,5) adapted to TPC-C for OLAP

Evaluation: DB X

	Q1	Q2	Q3	Q4	Q5
HyPer: C++ [ms]	142	374	141	203	1416
Compile time[ms]	1556	2367	1976	2214	2592
HyPer: LLVM [ms]	35	125	80	117	1105
Compile time[ms]	16	41	30	16	34
VectorWise [ms]	98	-	257	436	1107
MonetDB [ms]	72	218	112	8168	12028
DB X [ms]	4221	6555	6410	3830	15212

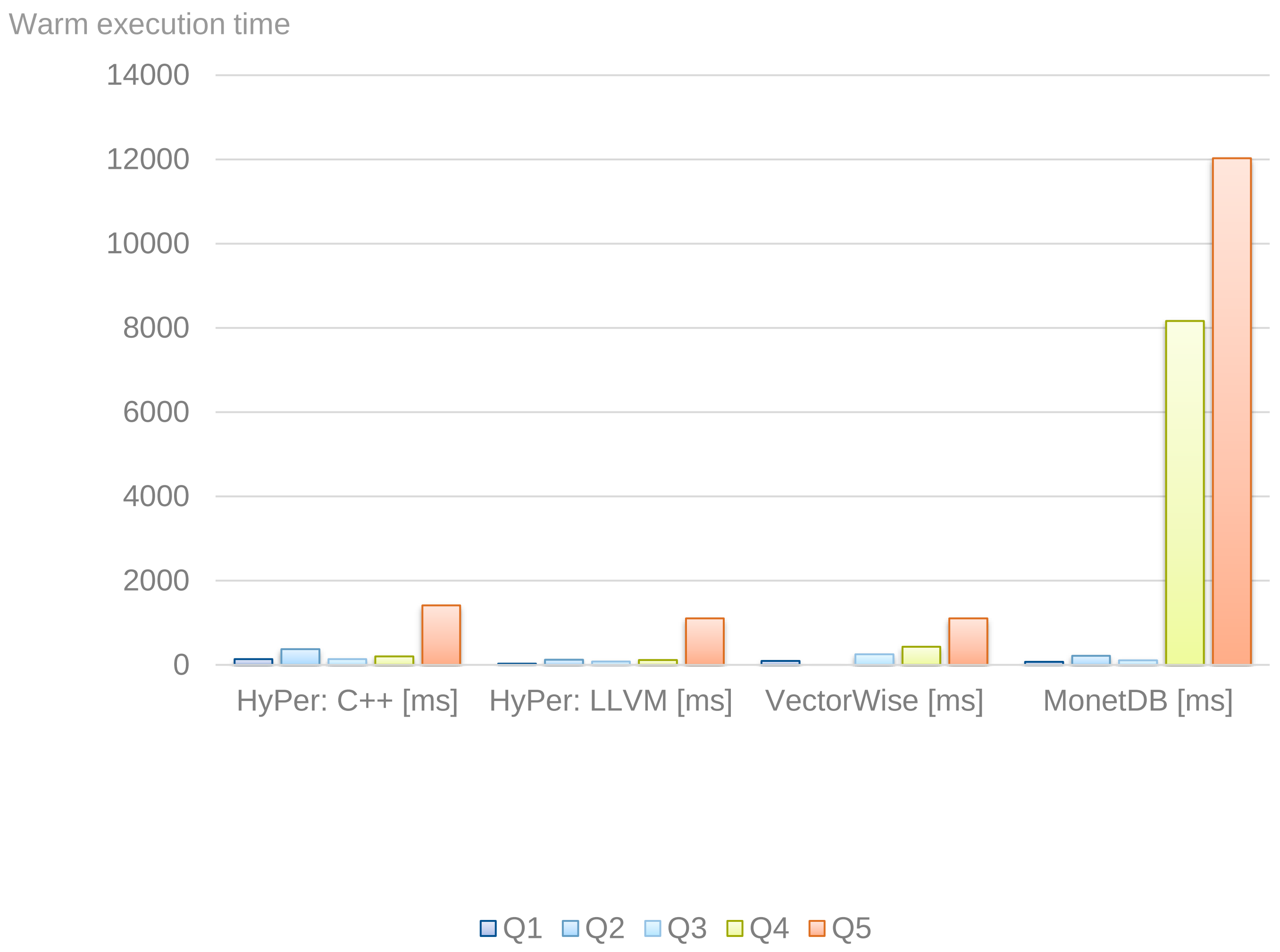
Evaluation: DB X



Evaluation: MonetDB

	Q1	Q2	Q3	Q4	Q5
HyPer: C++ [ms]	142	374	141	203	1416
Compile time[ms]	1556	2367	1976	2214	2592
HyPer: LLVM [ms]	35	125	80	117	1105
Compile time[ms]	16	41	30	16	34
VectorWise [ms]	98	-	257	436	1107
MonetDB [ms]	72	218	112	8168	12028
DB X [ms]	4221	6555	6410	3830	15212

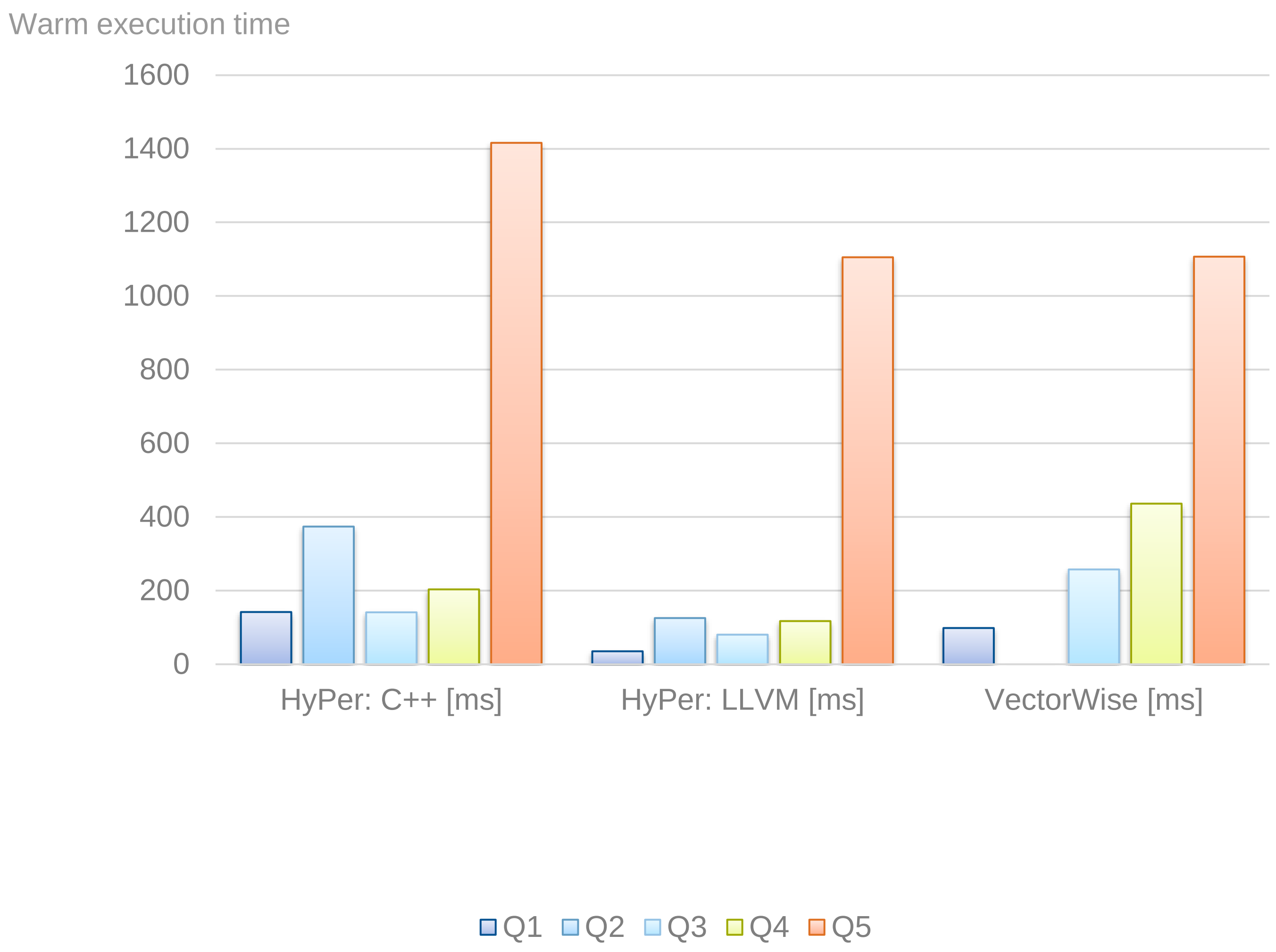
Evaluation: MonetDB



Evaluation: VectorWise

	Q1	Q2	Q3	Q4	Q5
HyPer: C++ [ms]	142	374	141	203	1416
Compile time[ms]	1556	2367	1976	2214	2592
HyPer: LLVM [ms]	35	125	80	117	1105
Compile time[ms]	16	41	30	16	34
VectorWise [ms]	98	-	257	436	1107
MonetDB [ms]	72	218	112	8168	12028
DB X [ms]	4221	6555	6410	3830	15212

Evaluation: VectorWise



Evaluation

➢ We implemented this in our HyPer system

- initially we generated C++ code from code fragments
- then, switched to the data-centric LLVM code: comparison C++ vs. LLVM

➢ Compared it with other systems

- MonetDB 1.36.5, Ingres VectorWise 1.0, DB X (commercial DBS)
- 5 TPC-H (Q1,2,3,4,5) queries adapted to TPC-C for OLAP

➢ All five queries (using the Callgrind Tool of Valgrind 3.6.0)

- MonetDB ⬄ LLVM version of HyPer

Evaluation: Branches and Cache Misses

	Q3		Q4		Q5	
	LLVM	MonetDB	LLVM	MonetDB	LLVM	MonetDB
branches	14,362,660	127,944,656	32,243,391	408,891,838	11,427,746	333,536,532
Mispredicts	696,839	1,884,185	1,182,202	6,577,871	639	6,726,700
I1 misses	791	386,561	508	290,894	490	2,061,837
D1 misses	2,341,531	7,557,629	3,480,437	20,981,731	776,417	8,573,962
L2d misses	1,420,628	5,947,845	3,424,857	17,072,319	776,229	7,552,794
I refs [mil]	208	944	282	3,140	159	2,089

Conclusion

- Data-centric query processing shows excellent performance

⇒ Minimize number of memory accesses

⇒ Data kept in CPU registers

⇒ Increases locality, reduces branching

- LLVM is an excellent tool for code generation

⇒ Fast on demand code generation

⇒ Good code quality

⇒ Portable and well maintained

⇒ Low compile times

Thank you for your attention !

Mohamed Attia, Patrick Zimmermann, Thomas Torggler