Why is performance productivity poor on modern architectures?

Dagstuhl Seminar on Petacomputing, Feb 13–17, 2006

Jan Treibig[1] Georg Hager[2]

[1]Lehrstuhl für Systemsimulation, FAU Erlangen-Nürnberg

[2]Regionales Rechenzentrum Erlangen, FAU Erlangen-Nürnberg

February 16, 2006

Contents

Application writers at new frontiers

- Parallelization techniques well established among HPC “power users”
- Don’t forget: Petacomputing == gigacomputing at the serial level!
- New users forced into parallelization with obstacles to take:
  - Multi-/many-core
  - ccNUMA
  - Network topologies
  - Efficient I/O
  - Strange architectures
  - Stupid compilers
  - Lack of mature and “simple” tools
  - . . . you name it!
- Interesting paradigm: “Constellation” clusters
  - Motivation: 30/60 TFlop/s SGI Altix 4000 to be installed in Bavaria very soon
  - Promising strategy: Choose the “easiest” path and use OpenMP if possible!

OpenMP?

- Incremental parallelism, serial equivalence
- Good language support, at least in theory
  - Lots of bugs, esp. in C++ compilers
- Easy to learn, hard to master
- Flexible enough to emulate “minimalistic MPI”
- DSM variants available (Cluster OpenMP . . .)
- Advanced tools for correctness checking
- Thread safety $\leftrightarrow$ performance issues?

The C++/OpenMP/ccNUMA mess

- C++ is all about objects and templates, and it should stay that way when doing parallel programming
- Problem: Constructors usually called in a serial region. Test case:

Example (Wrapped `double`)

```
class D {
  double d;
public:
  D(double _d=0.0) throw() : d(_d) {}
  inline D operator+(const D& o) throw() {
    return D(d+o.d);
  }
  [...]
};
[...]
  D* A = new array[20000];
```

The C++/OpenMP/ccNUMA mess

- C++ is all about objects and templates, and it should stay that way when doing parallel programming
- Problem: Constructors usually called in a serial region. Test case:

Example (Wrapped double)

```
class D {
  double d;
public:
  D(double _d=0.0) throw() : d(_d) {}
  inline D operator+(const D& o) throw() {
    return D(d+o.d);
  }
  [...]
};
[...]
  D* A = new array[20000]; // Locality problem!
```

The C++/OpenMP/ccNUMA mess

- C++ is all about objects and templates, and it should stay that way when doing parallel programming
- Problem: Constructors usually called in a serial region. Test case:

Example (Wrapped `double`)

```
class D {
  double d;
public:
  D(double _d=0.0) throw() : d(_d) {}
  inline D operator+(const D& o) throw() {
    return D(d+o.d);
  }
  [...]
};
[...]
  D* A = new array[20000]; // Locality problem!
```

- Solution: Use parallel “first touch” in allocation

Implementing first touch the easy way

- Correct placement by allocating with “first touch”:

Example (First touch allocation)

```
template <class T> T* pnew(size_t n) {
  [...]
  char *p = new char[len];
#pragma omp parallel for schedule(static) private(ofs)
    for(i=0; i<pages; ++i) {
      ofs = static_cast<size_t>(i) « PAGE_BITS;
      p[ofs]=0;
    }
    for(ofs=0; ofs<n; ++ofs) {
      new(static_cast<void*>(p+ofs*st)) T;

    }
  return static_cast<T*>(p);
}
```

Implementing first touch the easy way

- Correct placement by allocating with “first touch”:

Example (First touch allocation)

```
template <class T> T* pnew(size_t n) {
  [...]
  char *p = new char[len];
#pragma omp parallel for schedule(static) private(ofs)
    for(i=0; i<pages; ++i) {
      ofs = static_cast<size_t>(i) « PAGE_BITS;
      p[ofs]=0;
    }
    for(ofs=0; ofs<n; ++ofs) {
      new(static_cast<void*>(p+ofs*st)) T;
// placement new
    }
  return static_cast<T*>(p);
}
```

Implementing first touch with STL `vectors`

- But we want to do *real* C++ and use STL, in a user-friendly way.
- Solution: Design a NUMA-aware STL allocator

Example (STL NUMA allocator)

```
template <class T> class NUMA_Allocator {
public: [...]
  T* allocate(size_type n, const void *lH=0) {
    size_type ofs,len = n*sizeof(T);
    char *p = malloc(len);
#pragma omp parallel for schedule(static) private(ofs)
    [ ... same as before ...]
  }
  void construct(T* p, const T& x) {
    new(p) value_type(x);
  }
  void destroy(T* p) { p->~T(); }
};
```

Getting bad performance the easy way

- Now we can do
  `vector<double,NUMA_Allocator<double> > A(20000);`
  and use A in OpenMP loops and be happy. Or can't we?
- Performance penalties of 1-thread vector triad with respect to "vanilla" version for out-of-cache data set:

	S,op	S,it	S,op,O	S,it,O	d,O
Intel V9 IA64	0.50	0.99	0.25	0.28	0.98
Intel V9 EM64T	0.78	0.80	0.64	0.79	1.00
PGI x86_64	0.53	0.90	0.47	0.68	0.79
Pathscale x86_64	0.87	0.87	0.81	0.87	1.00
MIPSPro MIPS	0.78	1.00	0.84	0.95	1.00

Legend: S=STL, op=`operator[]`, it=`iterator`, O=OpenMP

Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction

Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`

Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.

Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?

Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
- Generally speaking, OpenMP is bad for serial performance

Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
- Generally speaking, OpenMP is bad for serial performance
  - Not only in cache, but also for streaming

Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
- Generally speaking, OpenMP is bad for serial performance
  - Not only in cache, but also for streaming
  - Some issues with NT stores on x86, but in general?

Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
- Generally speaking, OpenMP is bad for serial performance
  - Not only in cache, but also for streaming
  - Some issues with NT stores on x86, but in general?
- Should we start from scratch and dump the shared memory approach altogether?

Lessons learned from the high-level approach

- Compilers are extremely sensitive to any obstruction
  - There is only one layer between `vector<T>::operator[]` and `vector<T>::iterator`
- STL was designed to expose all necessary code to the compiler, enabling “vanilla-like” optimization. It seems that this approach failed.
- Should we abandon STL, or even C++, for low-level operations?
- Generally speaking, OpenMP is bad for serial performance
  - Not only in cache, but also for streaming
  - Some issues with NT stores on x86, but in general?
- Should we start from scratch and dump the shared memory approach altogether?
- Are we tackling tomorrow’s performance challenges with yesterday’s tools?

Getting good performance – the hard way

Observation:

The memory wall is not the only problem we are facing. After algorithmic and data layout changes the mapping of the high level language to the ISA is an important issue.

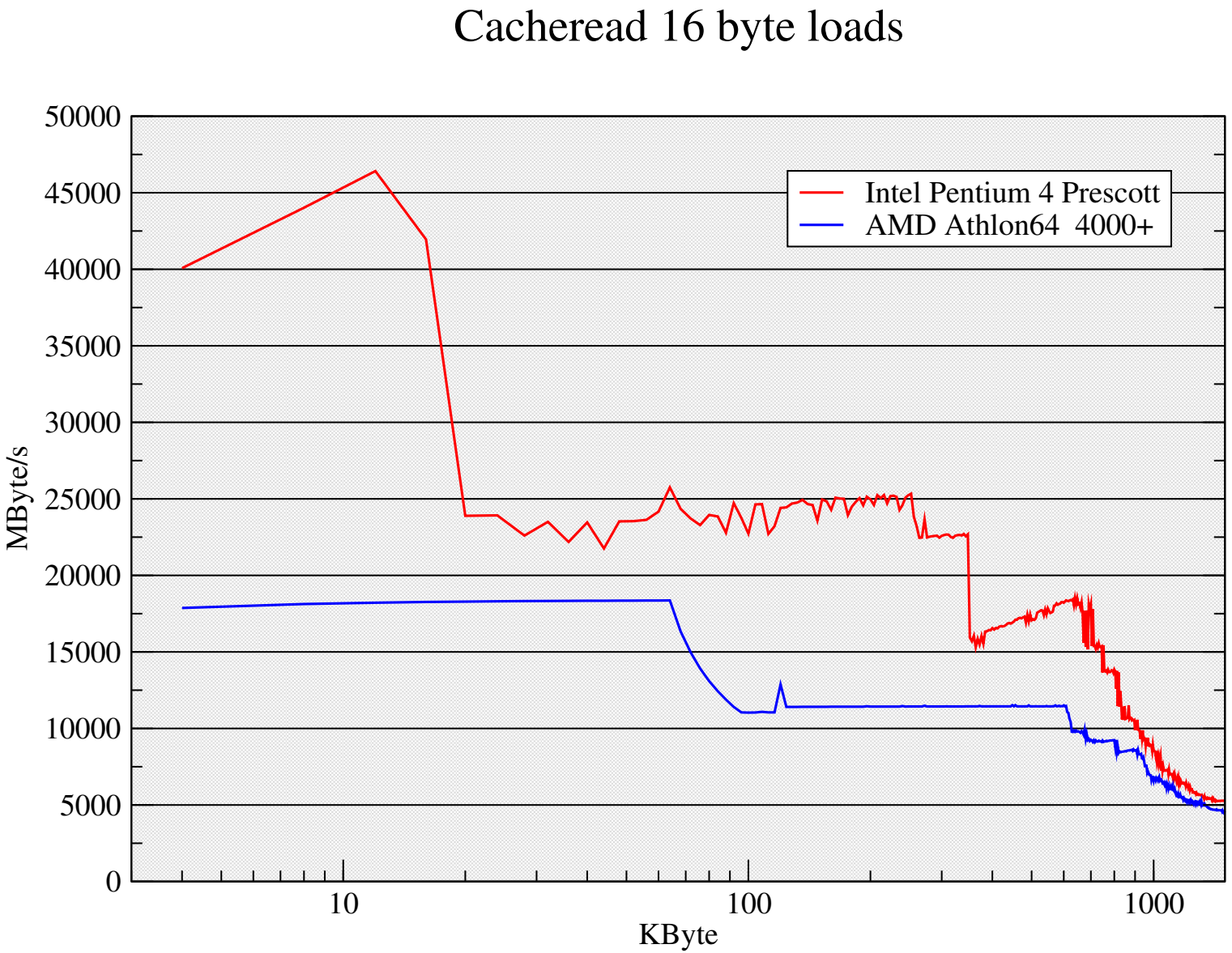
Compilers have difficulties to utilize the performance of modern CPUs. Reasons are:

- Still the well-known issue that caches are not transparent with regard to performance
- Developements in modern CPU architectures:
  - Prefetching
  - SIMD
  - Special instructions: e.g. Non temporal stores
- General code quality (address calculation, register scheduling)

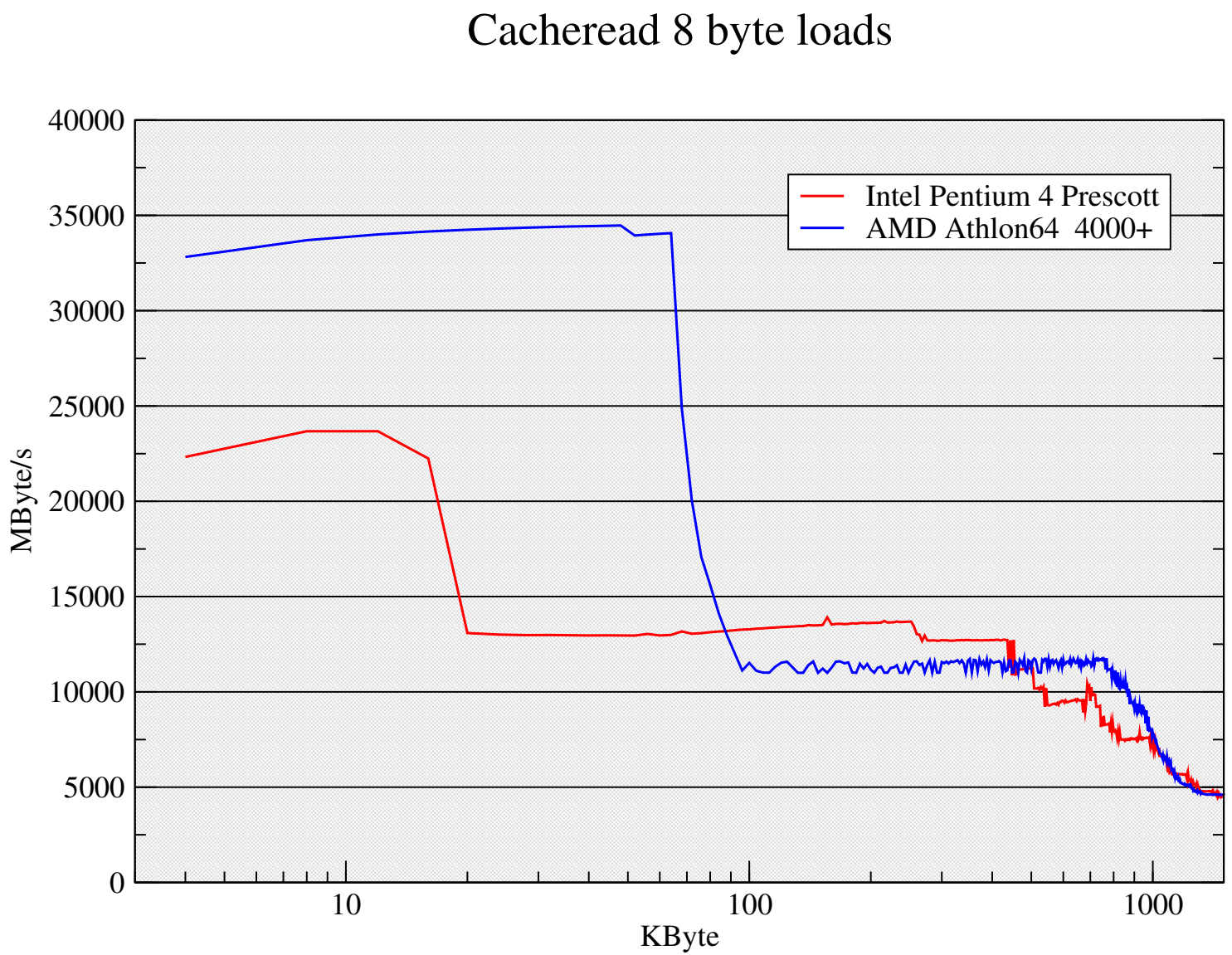
Architectural Overview

	Intel Prescott	**AMD Athlon64**
Clock speed	3.2 GHz	2.4 GHz
Cacheline length	64(128) Byte	64 Byte
L2 Cache Size	1 MByte	1 MByte
L1 Cache Size	16 kByte	64 Byte
L2 Latency	56 cycles (min 21)	13 cycles (min 11)
L2 Read Bandwidth	23 GB/s	13 GB/s
L2 Write Bandwidth	12 GB/s	12 GB/s
L1 Latency	4 cycles (min 1)	3 cycles (min 2)
L1 Read Bandwidth	46 GB/s	35 GB/s
L1 Write Bandwidth	12 GB/s	35 GB/s
Memory Read Bandwidth	5.8 GB/s	6.1 GB/s
Memory Write Bandwidth	4.1 GB/s	6.1 GB/s

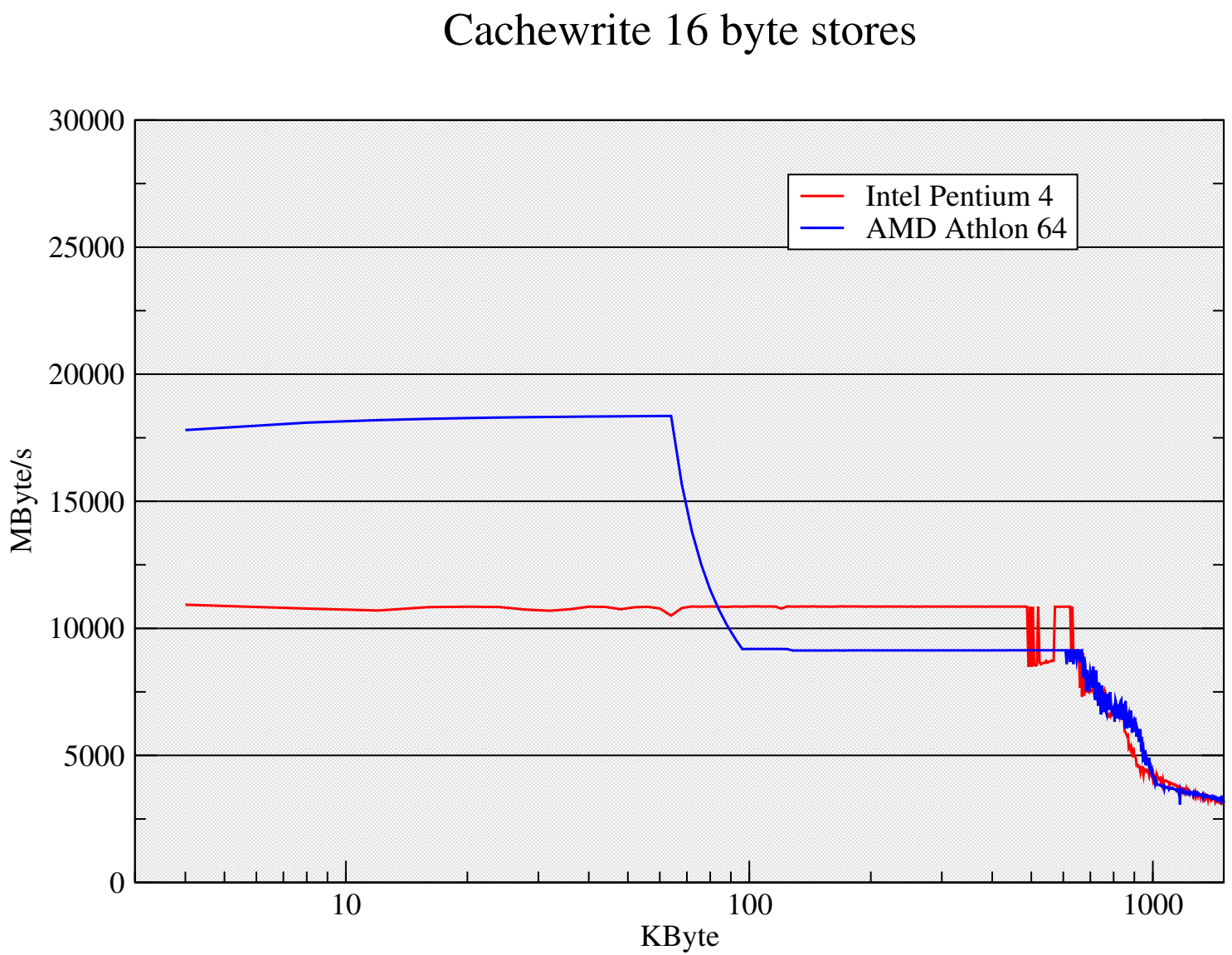
Memory Hierarchy Cache Bandwidth



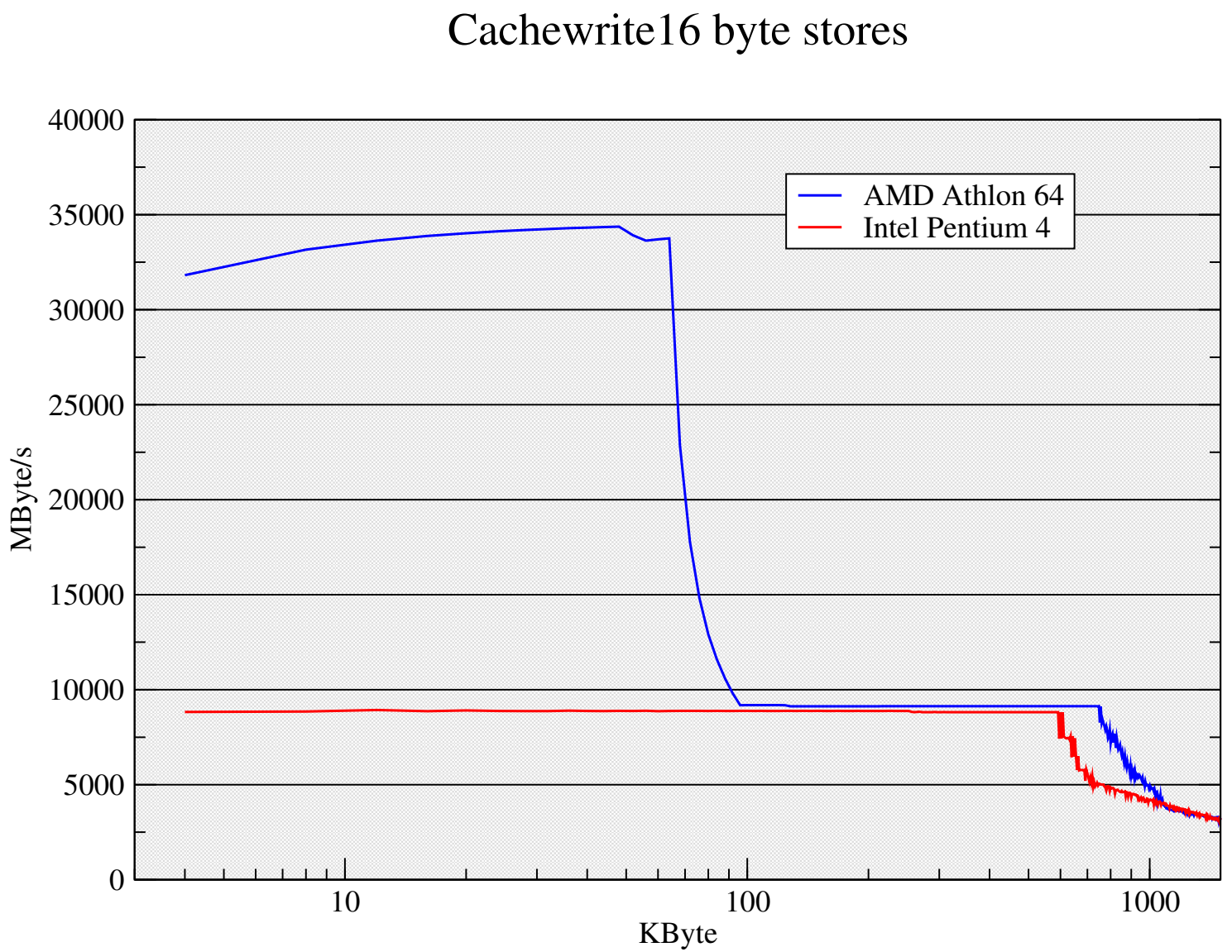
Memory Hierarchy Cache Bandwidth



Memory Hierarchy Cache Bandwidth



Memory Hierarchy Cache Bandwidth



Peak Performance

	Pentium 4	**Athlon64**
Add	2920 Mflops (44.6 %)	2338 Mflops (48.7 %)
	23.3 GByte/s	18.7 GByte/s
MultAdd		
Add 2		
Add 2 var		

Peak Performance

	Pentium 4	**Athlon64**
Add	2920 Mflops (44.6 %)	2338 Mflops (48.7 %)
	23.3 GByte/s	18.7 GByte/s
MultAdd	5229 MFlops (81 %)	4153 MFlops (86 %)
	20.9 GByte/s	16.1 GByte/s
Add 2		
Add 2 var		

Peak Performance

	Pentium 4	Athlon64
Add	2920 Mflops (44.6 %)	2338 Mflops (48.7 %)
	23.3 GByte/s	18.7 GByte/s
MultAdd	5229 MFlops (81 %)	4153 MFlops (86 %)
	20.9 GByte/s	16.1 GByte/s
Add 2	2339 MFlops (36 %)	1187 (24 %)
	37.4 GByte/s	19.0 GByte/s
Add 2 var		

Peak Performance

	Pentium 4	Athlon64
Add	2920 Mflops (44.6 %)	2338 Mflops (48.7 %)
	23.3 GByte/s	18.7 GByte/s
MultAdd	5229 MFlops (81 %)	4153 MFlops (86 %)
	20.9 GByte/s	16.1 GByte/s
Add 2	2339 MFlops (36 %)	1187 (24 %)
	37.4 GByte/s	19.0 GByte/s
Add 2 var	2454 MFlops (38 %)	2082 MFlops (43.3 %)
	39.2 GByte/s	33.3 GByte/s

```
movdqa xmm1, [x+ecx*8]
movdqa xmm3, [y+ecx*8]
addpd xmm3, xmm1
replaced by
movdqa xmm4, [x+ecx*8]
addpd xmm4, [y+ecx*8]
```

Peak Performance: The Code

Example (Peakflop Code snippet)

```
.loop:
movapd    xmm1, [x+ecx*8]
addpd     xmm6, xmm0
mulpd     xmm1, xmm7
movapd    xmm2, [x+ecx*8+16]
addpd     xmm5, xmm0
mulpd     xmm2, xmm7
movapd    xmm3, [x+ecx*8+32]
addpd     xmm1, xmm0
mulpd     xmm3, xmm7
movapd    xmm4, [x+ecx*8+48]
addpd     xmm2, xmm0
mulpd     xmm4, xmm7
add ecx, 8
cmp ecx,1000
jb .loop
```

Intel vs. AMD

- Low latency (AMD) against high bandwidth (Intel)
- Intel is more sensitive against type of instructions
- AMD suffers from low bandwidth L2 Cache connection
- For streaming applications the Netburst Architecture is superior
- AMD has very good memory connection
- Hardware prefetcher works more efficently on the P4
- Software prefetch instructions work more efficently on the Athlon64

Memcpy: Influence of instruction types

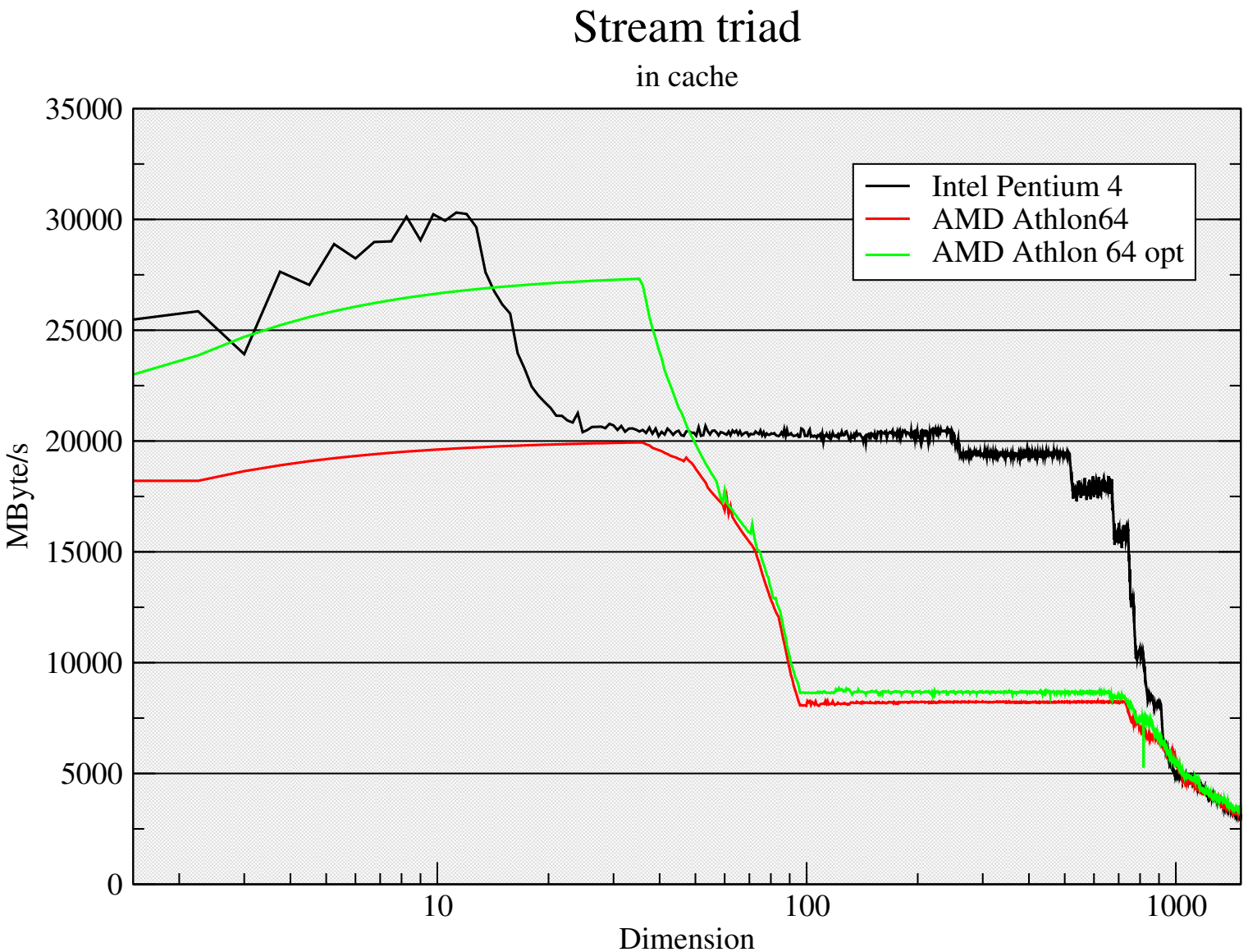
	Pentium 4	**Athlon64**
CISC	2481 MB/s	2001 MB/s
RISC	2424 MB/s	2834 MB/s
MMX	2489 MB/s	2880 MB/s
MMX NT	3737 MB/s	4104 MB/s
MMX NT SW-Prefetch	3964 MB/s	5199 MB/s
SSE NT SW-Prefetch	4012 MB/s	5206 MB/s
SSE2 Block Prefetch	4644 MB/s	6030 MB/s

Stream Triad: In Memory

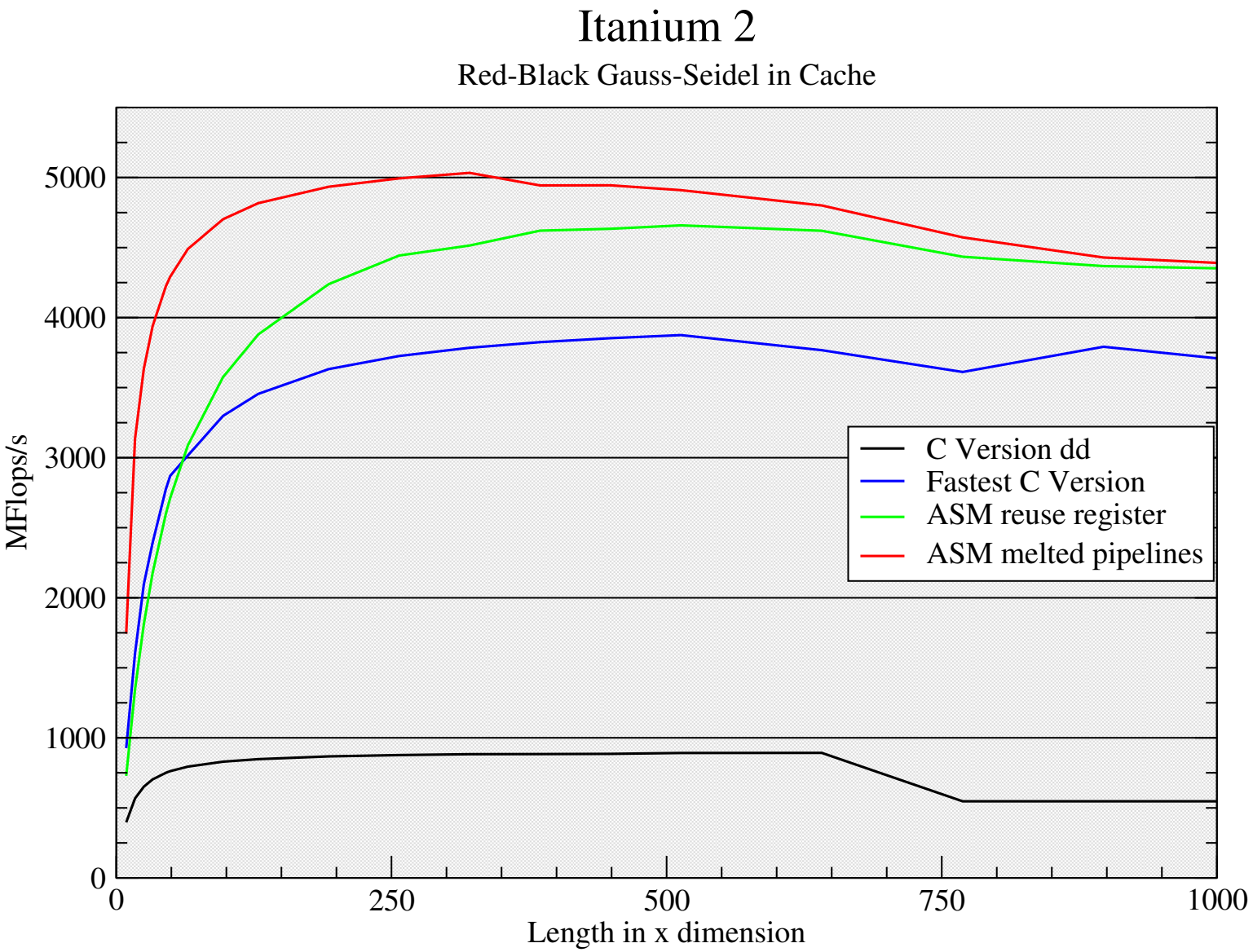
	Pentium 4	Athlon64
Compiler	4193 MB/s	4533 MB/s (3114 MB/s)
Optimized	4946 MB/s	5626 MB/s

The difference in performance is caused by effective prefetching and the seperation of prefetching data into cache and doing the actual computations with storing it back.

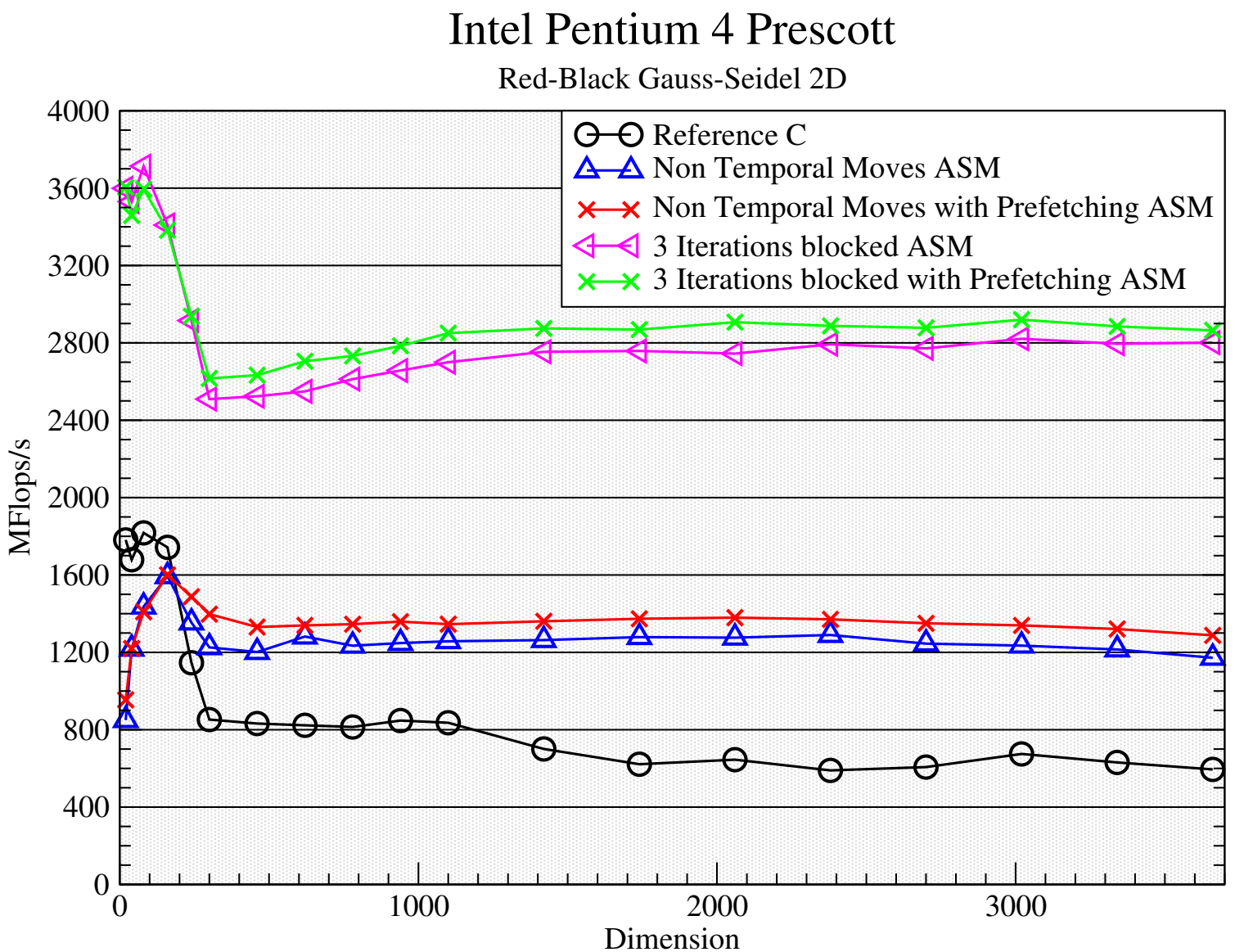
Stream Triad: In Cache



Software Pipelined Loops



Can you afford to waste a factor of 2-5?



Conclusion

- A gap is opening between hardware techniques and the capabilities of compilers
- Modern CPUs often are intransparent to the programmer
- Much performance is wasted by an inappropriate usage of the instruction set

Points for discussion

There is obviously a strong need for a tighter integration of ISA and Software. In addition to that the implementation of the ISA should be more transparent and reliable.

What can be done to solve that problem on hardware and software side?