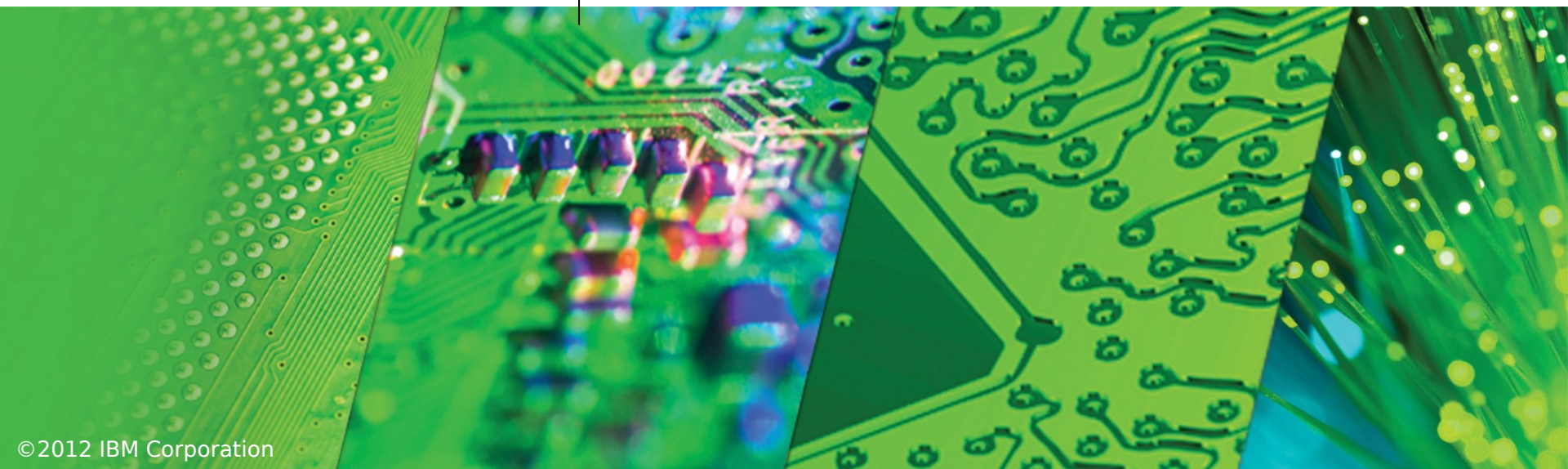
2012
IBM System z Technical University

Enabling the infrastructure for smarter computing

GCC enhancements for Linux on System z

zLG14

Mario Held



Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Oracle and Java are registered trademarks of Oracle and/or its affiliates in the United States, other countries, or both.

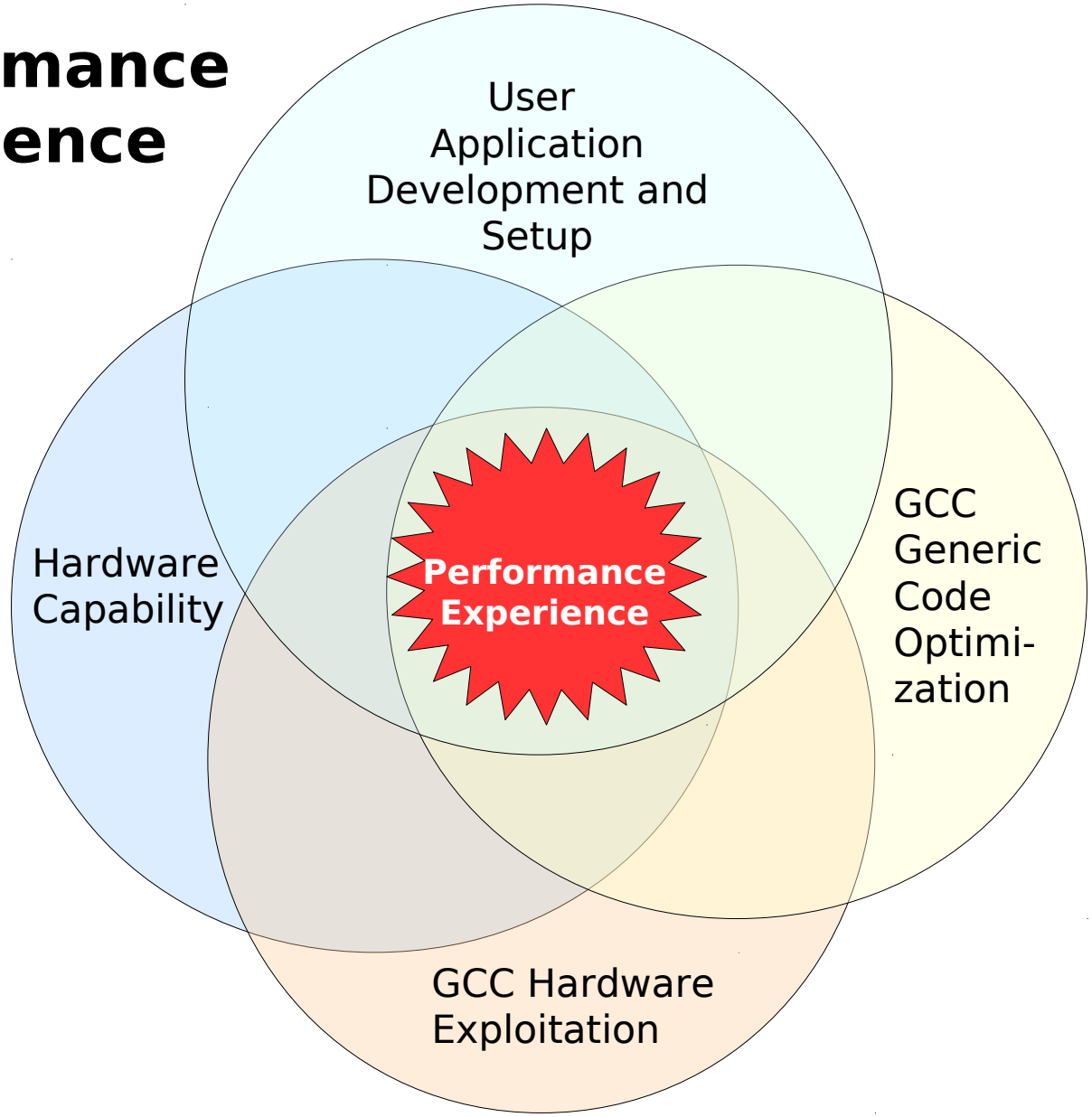
Red Hat, Red Hat Enterprise Linux (RHEL) are registered trademarks of Red Hat, Inc. in the United States and other countries.

SLES and SUSE are registered trademarks of Novell, Inc., in the United States and other countries.

SPEC and the "performance chart" SPEC logo are registered trademarks of the Standard Performance Evaluation Corporation.

Other product and service names might be trademarks of IBM or other companies.

Performance Experience



Agenda

- **Hardware Capability of the z196 and zEC12**
- GCC Generic Code Optimization
- GCC Hardware Exploitation

z196 PU core

- Each core is a super-scalar, out of order processor with these characteristics:
  - ✓ Six execution units
    - 2 fixed point (integer), 2 load/store, 1 binary floating point, 1 decimal floating point
  - ✓ Can decode up to three instructions per cycle
  - ✓ Can execute up to five instructions per cycle
  - ✓ Runs at 5.2 GHz
  - ✓ Each core has 3 private caches
    - 64KB L1 cache for instructions, 128KB L1 cache for data
    - 1.5MB L2 cache containing both instructions and data
  - ✓ Approximately 100 new instructions
  - ✓ Execution can occur out-of-(program)-order

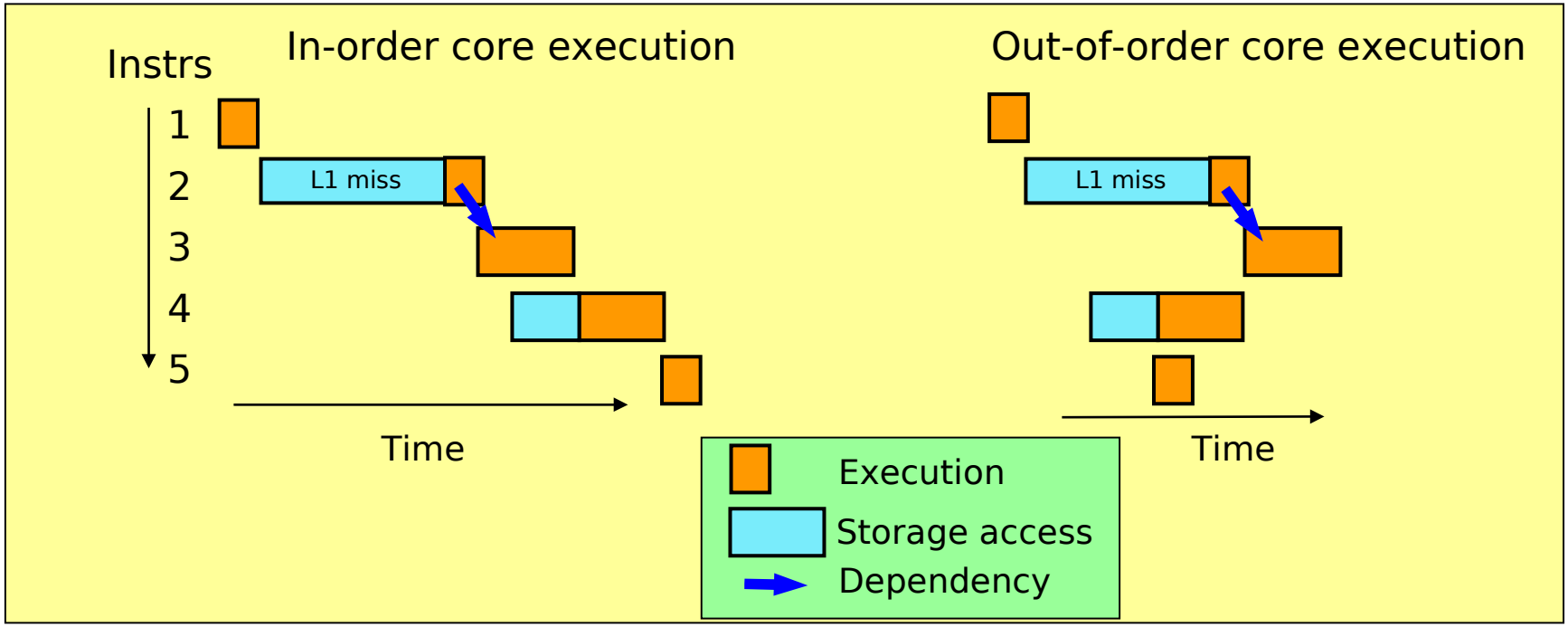
Out of Order detail

Out of order yields significant performance benefit through

- Re-ordering instruction execution
  - ✓ Instructions stall in a pipeline because they are waiting for results from a previous instruction or the execution resource they require is busy
  - ✓ In an in-order core, this stalled instruction stalls all later instructions in the code stream
  - ✓ In an out-of-order core, later **independent** instructions are allowed to execute ahead of the stalled instruction
- Re-ordering storage accesses
  - ✓ Instructions which access storage can stall because they are waiting on results needed to compute storage address
  - ✓ In an in-order core, later instructions are stalled
  - ✓ In an out-of-order core, later storage-accessing instructions which can compute their storage address are allowed to execute
- Hiding storage access latency
  - ✓ Storage accesses can miss the caches and require 10 to 500 additional cycles to retrieve the storage data
  - ✓ In an in-order core, later instructions in the code stream are stalled
  - ✓ In an out-of-order core, later instructions which are not dependent on this storage data are allowed to execute

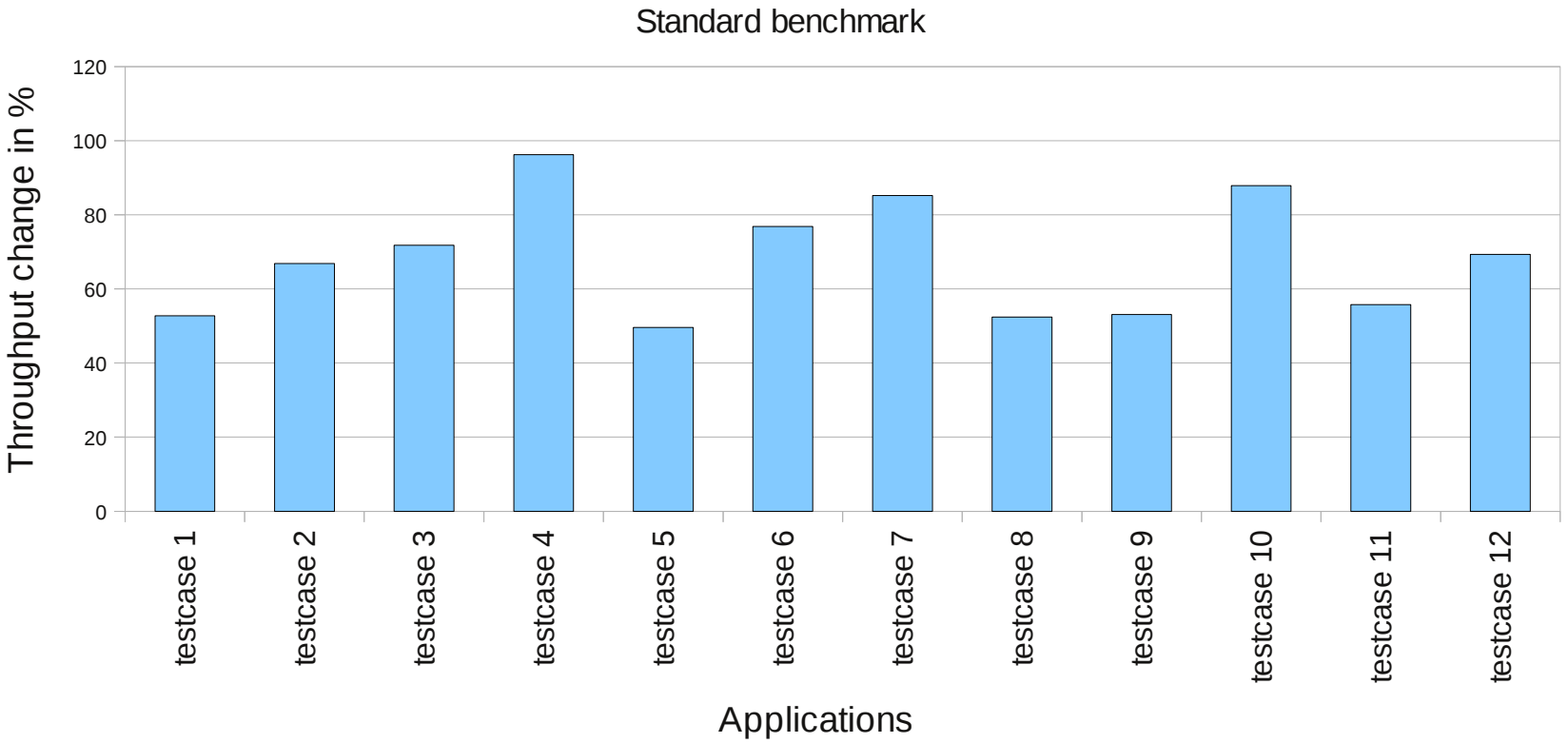
z196 Out-of-Order (OOO) benefit

- OOO yields significant performance benefit for compute intensive applications through
  - ✓ Re-ordering instruction execution
  - ✓ Later (younger) instructions can execute ahead of an older stalled instruction
  - ✓ Re-ordering storage accesses and parallel storage accesses
- OOO maintains good performance growth for traditional applications



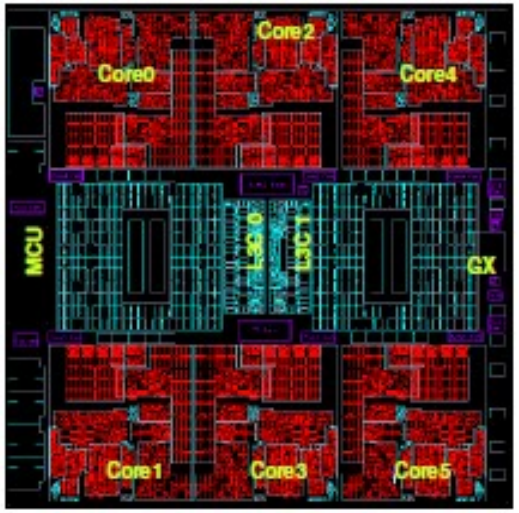
Compare performance using the same workload on z196 and z10

- Industry standard benchmark on RHEL6.2 with parameters -O3 -march=<z10 | z196> -funroll-loops
  - ✓ Improvement caused by hardware but also compiler adaption
  - ✓ Average throughput change is approximately 70 % with this benchmark

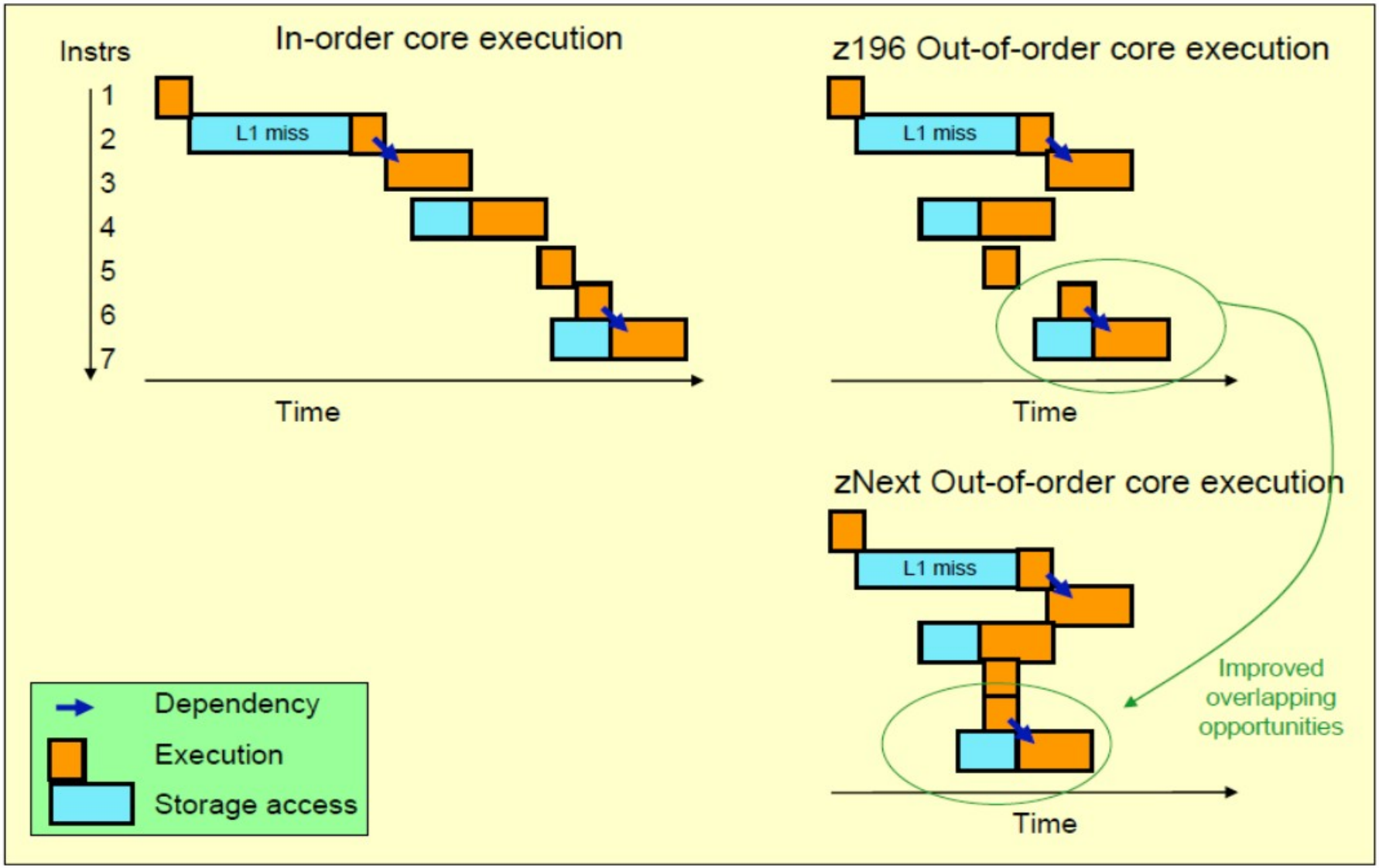


zEC12 PU core

- Each core is a super-scalar, out of order processor with these characteristics:
  - ✓ Six execution units
    - 2 fixed point (integer), 2 load/store, 1 binary floating point, 1 decimal floating point
  - ✓ Up to three instructions decoded / completed per cycle
  - ✓ Up to seven instructions issued per cycle
  - ✓ Runs at 5.5 GHz
  - ✓ Each core has 4 private caches
    - 64KB L1 cache for instructions, 96KB L1 cache for data
    - L2 cache
      - 1MB for instructions, 1MB for data
  - ✓ Better branch prediction
  - ✓ Enhanced out-of-(program)-order (OOO+) capabilities

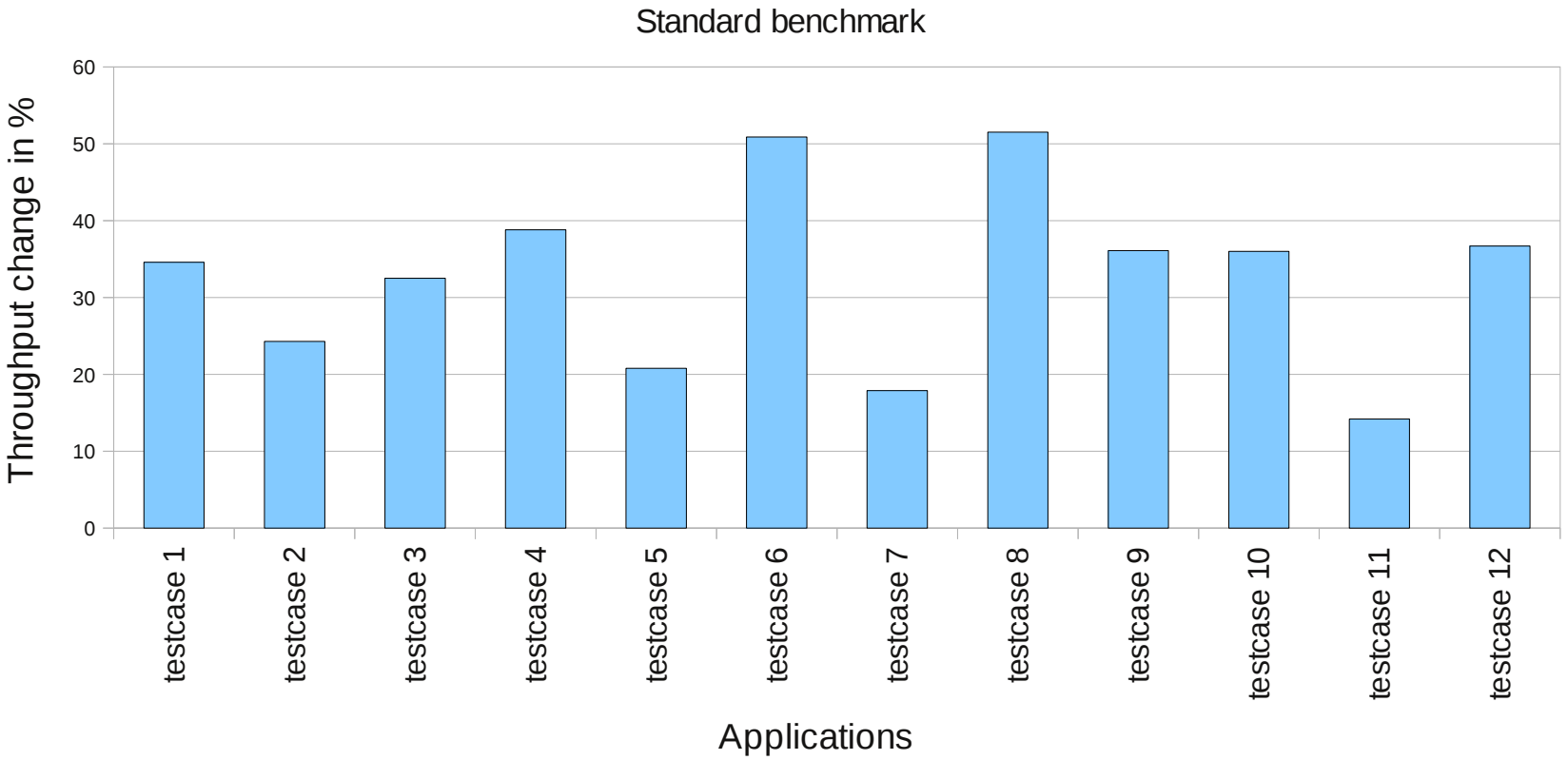


zEC12 Out-of-Order improved (OOO+) benefit



Compare performance using the same workload on zEC12 and z196

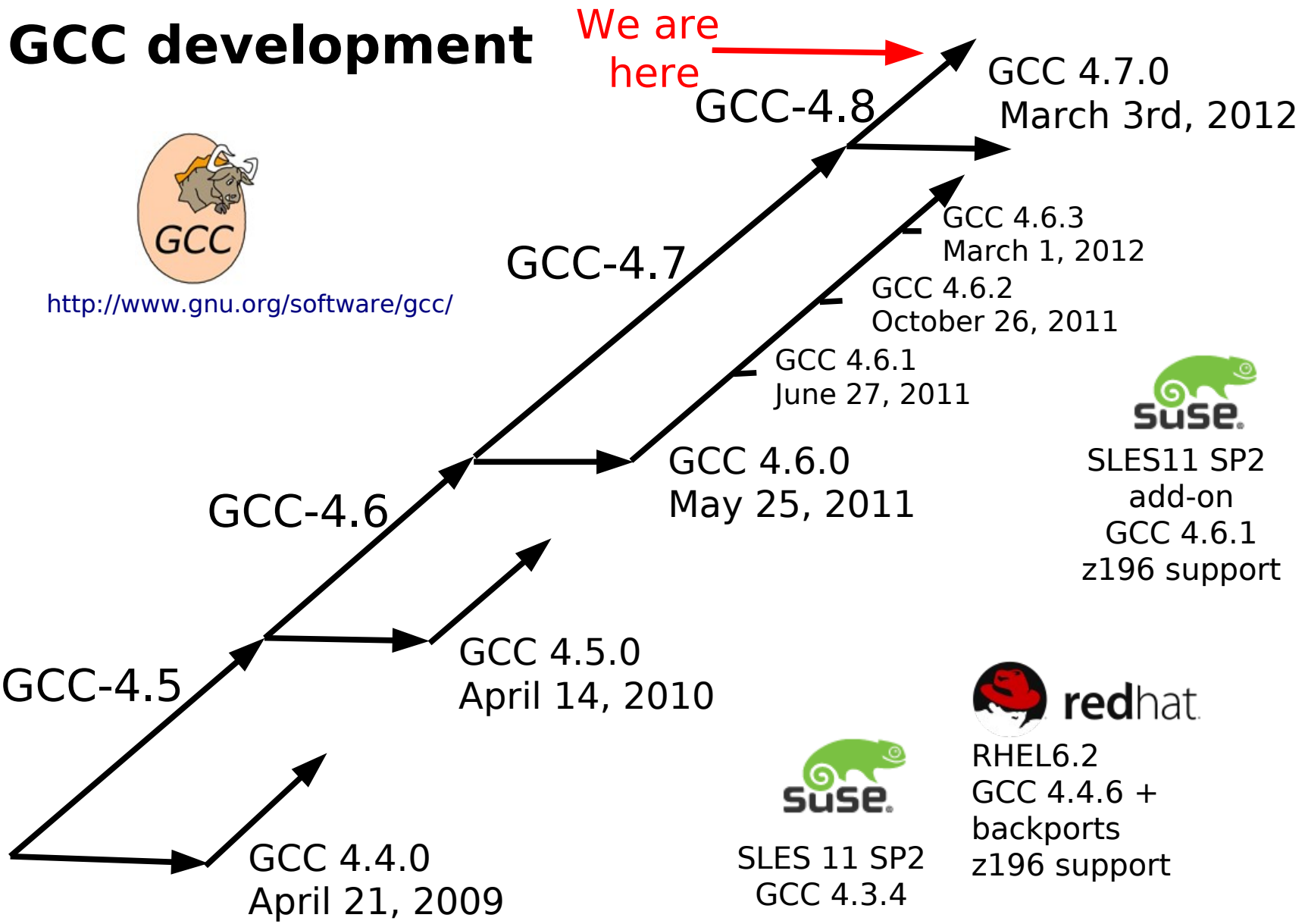
- Industry standard benchmark on RHEL6.2 with parameters -O3 -march=z196 -funroll-loops
  - ✓ Improvement caused by hardware only at this point in time
  - ✓ Average throughput change is approximately 30 % with this benchmark



Agenda

- Hardware Capability of the z196
- **GCC Generic Code Optimization**
- GCC Hardware Exploitation

GCC development



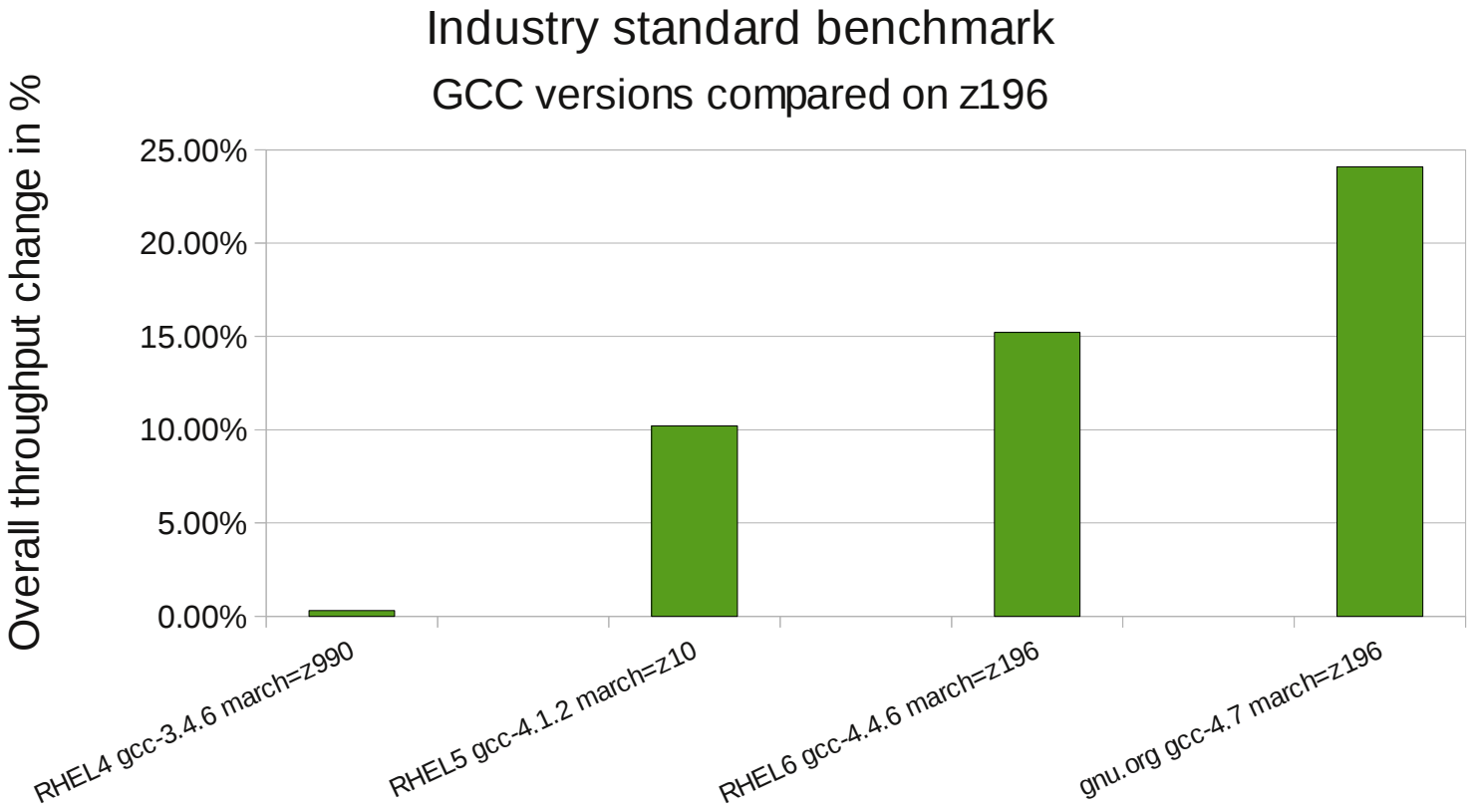
GCC versions in Linux on System z supported distributions

GCC stream	x.y.0 release	Included in SUSE distribution	Included in Red Hat distribution
GCC-3.3	05/2003	SLES9 (z990 backport)	
GCC-3.4	04/2004		RHEL4 (z990 support)
GCC-4.0	04/2005		
GCC-4.1	02/2006	SLES10 (z9-109 support)	RHEL5 (z9-109 support)
GCC-4.2	05/2007		
GCC-4.3	05/2008	SLES11 (z10 backport)	
GCC-4.4	04/2009		RHEL5.6**/6.1 (z196 backport)
GCC-4.5	04/2010	SLES11 SP1	
GCC-4.6	03/2011	SLES11 SP2 (z196 support)*	
GCC-4.7	03/2012	?	?

* included in SDK, optional, not supported
** fully supported add-on compiler

GCC Evolution

- Comparing RHEL4 – RHEL5 – RHEL6- gcc-4.7 (Feb 2012) on a z196
- Improved machine / instruction support, instruction scheduling
- Advantages of using current compilers are significant



GCC 4.7.0 Release – March 3rd 2012

- GCC celebrated 25th anniversary with the release of GCC 4.7.0
- GCC 4.7.0 common code highlights:
  - ✓ LTO improvements: reduced memory usage, better scalability
  - ✓ Inline heuristic improvements: known value optimizations will be taken into account when making inline decisions
  - ✓ String length optimization pass: string length tracking leads to more strcpy, strcat calls optimized to memcpy
  - ✓ OpenMP 3.1 standard implemented
  - ✓ More C++11 standard features (atomics)
  - ✓ C and C++ software transactional memory support has been merged!
- GCC 4.7.0 for Linux on System z:
  - ✓ System z specific improvements to the string length optimization pass
  - ✓ Improved loop optimization

GCC Link Time Optimization

- Link Time Optimization (LTO) enables cross-module optimizations without changing the build infrastructure.
- Problem: Current build mechanics pass one source code file at a time to the compiler → cross-module optimizations not possible.
- Solution: Optimization is postponed until link-step when all the required modules are known.
- LTO compilation procedure:
  - ✓ First the compilation units are optimized separately
  - ✓ GCC internal code representation (GIMPLE) is embedded into the object file
  - ✓ During link-step the objects are passed to the compiler again
  - ✓ Compiler uses the embedded information to redo the optimization step
- High potential together with profile directed feedback (PDF).
- GCC support introduced with GCC 4.5 and matured since then.
- First experiments already show a single-digit performance improvement with LTO alone (no System z specific tuning so far).
- Options to add: -flto -fwhole-program

GCC Feedback Driven Optimization (FDO)

- FDO is also known as Profiled Directed Feedback (PDF)
- With FDO the compile is done in three phases:
  - ✓ Profile code generation, instrumentation code gets inserted
  - ✓ Training run while statistic information gets collected into a file, especially which code parts are used how often
  - ✓ Feedback optimization using the collected data from the previous phase to guide the optimization routines for instance for branch prediction or loop unrolling
- FDO produces in most cases significant better code and improves performance significantly
- FDO requires more compile time because of compiling twice and doing the test run, but it is usually worth the investment
- Best results if the codes' hot paths are not depending on the single input data
  - ✓ The advantage of FDO depends on a really representative training workload
  - ✓ If the training workload is not good your application could even run more slowly
- Option to add in the first pass: -fprofile-generate
- Option to add in the second pass: -fprofile-use

Agenda

- Hardware Capability of the z196
- GCC Generic Code Optimization
- **GCC Hardware Exploitation**

GCC Compile options

- Produce highly optimized code
  - ✓ Options -O3 or -O2 (often found in delivered Makefiles) are a good starting point and best tested by IBM
  - ✓ Choose the parameter -march = <e.g. z9-109, z10, z196> matching the oldest by your program supported target machine
    - The parameter -march determines the instruction set and defines the same value to the -mtune value implicitly
    - **Hint: -march is only upward compatible**
    - If your new hardware is not supported by the distribution's default compiler there may be a newer optional compiler on the distribution's SDK which is worth to give it a try
  - ✓ Optimize GCC instruction scheduling with the frequently used target machine in mind using parameter -mtune = <e.g. z9-109, z10, z196>
- Fine Tuning: additional general options on a file-by-file basis
  - ✓ -funroll-loops has often advantages on System z
    - Unrolling is internally delimited to a reasonable value by default
  - ✓ Use of inline assembler for performance critical functions may have advantages
  - ✓ -ffast-math speeds up calculations (if not exact implementation of IEEE or ISO rules/specifications for math functions is needed)

zEnterprise z196 GCC support

Re-compiled code / applications get further performance gains through approximately 100 new instructions, e.g.:

- Load/Store-on-Condition Facility (6 new instructions)
  - ✓ Load or store conditionally executed based on condition code
  - ✓ Dramatic improvement in code with highly unpredictable branches
- Interlocked-Access Facility (12 new instructions)
  - ✓ Interlocked (atomic) load, value update and store operation in a single instruction
- Distinct-Operands Facility (22 new instructions)
  - ✓ Independent specification of result register (different than either source register)
  - ✓ Reduces register value copying
- Population-Count Facility (1 new instruction)
- Hardware implementation of bit counting ~5x faster than prior software implementations

zEnterprise zEC12 GCC support

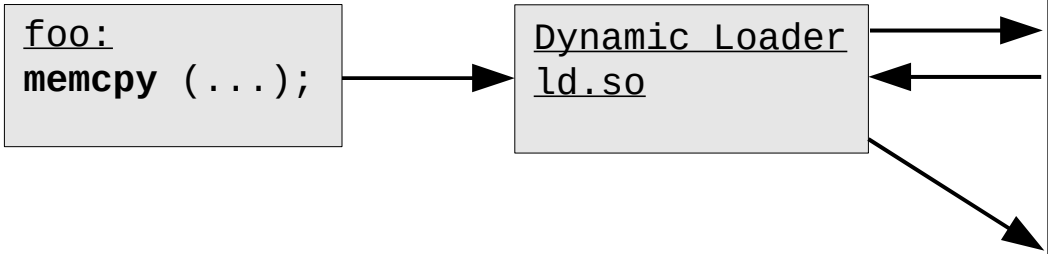
- zEC12 comes with new instructions
  - ✓ Transactional Memory support
  - ✓ Improved branch instructions
- New zEC12 instructions not included in supported Linux distributions yet
  - ✓ Best use of a zEC12 at this point in time with -march=z196
  - ✓ Ongoing development in GCC project at gcc.gnu.org

Per function tuning with STT_GNU_IFUNC support

- Problem: Compiler optimizations using new instructions get into the field very late. ISVs have to use -march=x with x being the oldest supported machine.
- Improvement: Provide critical functions optimized for different CPU levels and choose at runtime.
  - ✓ STT_GNU_IFUNC is a new ELF symbol type.
  - ✓ IFUNC symbols are resolved through a user provided resolver function.
  - ✓ Resolver function is invoked by the dynamic loader at runtime (only once).

STT_GNU_IFUNC implementation

- Usually a symbol is resolved by the dynamic linker using information provided in the symbol table of the involved objects
- STT_GNU_IFUNC introduces that this reference can be made upon other criteria. For instance upon the capabilities of the CPU where the code gets executed
  - ✓ Usually a symbol entry points to the symbol location directly
  - ✓ STT_GNU_IFUNC symbols instead point to a 'resolve' function
  - ✓ The 'resolve' function returns the symbol location that should be used for references to the symbol in question
- A simple example shows the implementation of various memcpy function instances



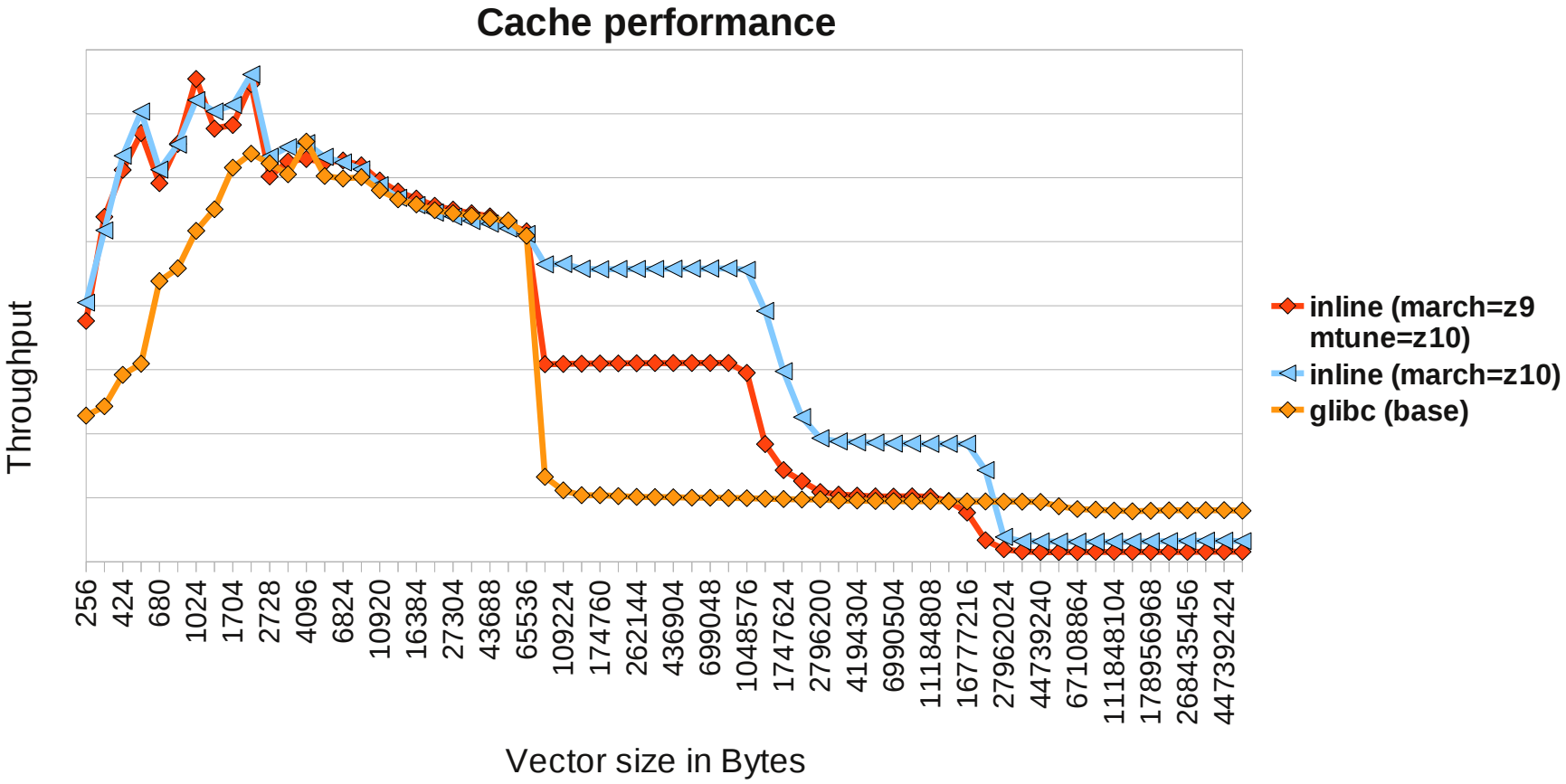

```
glibc:
resolve_memcpy () {
  if (ask_stfle () == z9)
    return &memcpy_z9;
if (ask_stfle () == z10)
    return &memcpy_z10;
if (ask_stfle () == z196)
    return &memcpy_z196;
  else
    return &memcpy_z196; }

memcpy_z10 (...) {
  pfd ...
}

memcpy_z196 ...
```

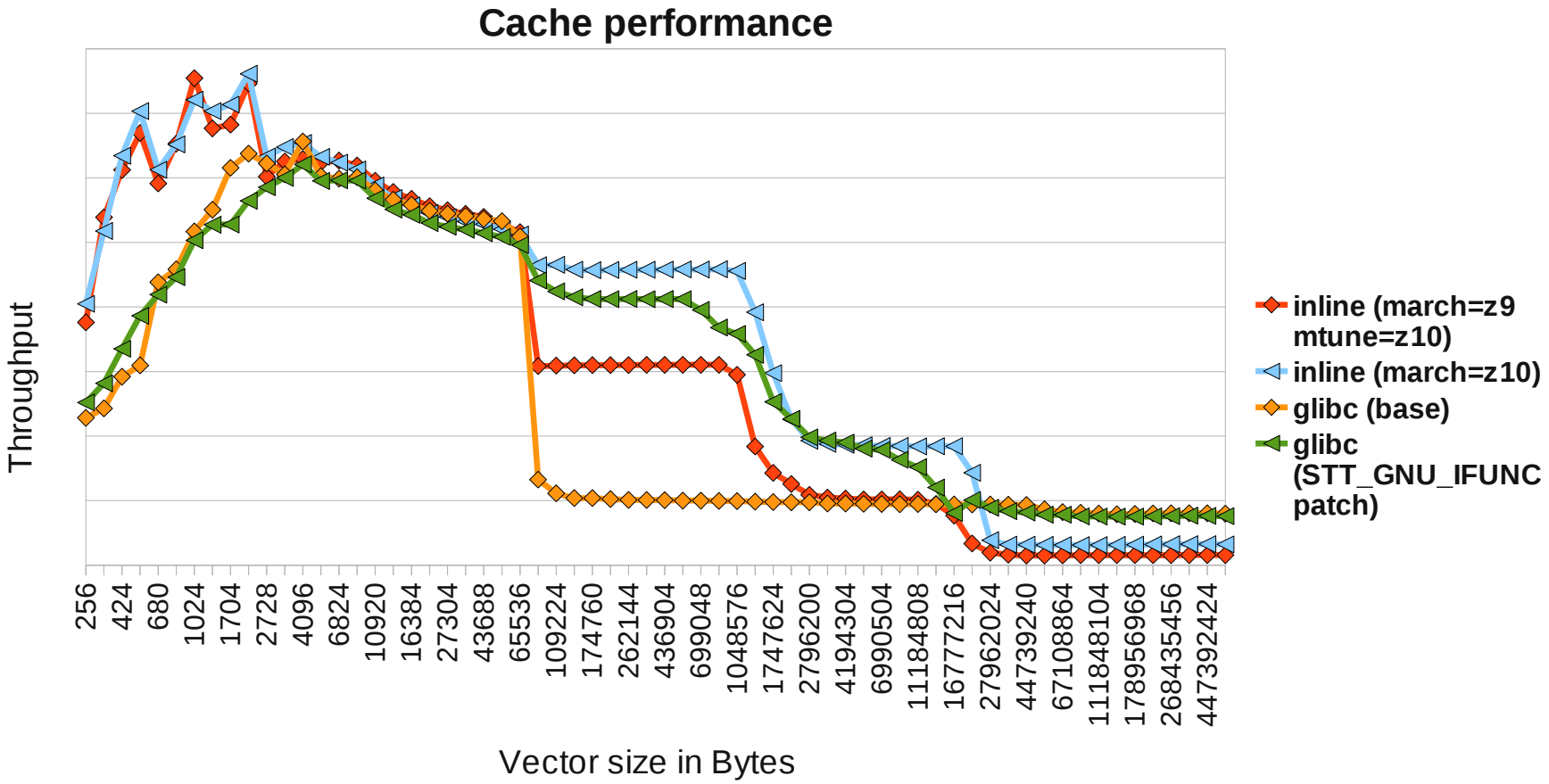
pfd is a prefetch instruction available on z10 and higher

STT_GNU_IFUNC - memcpy on z10



- STT_GNU_IFUNC results are expected between inlined code for z9 optimized (red) and z10 optimized (blue, target machine) results and better than old glibc implementation (yellow)

STT_GNU_IFUNC - memcpy on z10 (cont.)



- STT_GNU_IFUNC (green graph) expectation met with vector sizes > 2k
- Vector sizes <= 2K are faster when inlined (GCC adaptions upstream)
- Tremendous performance improvement compared to old glibc implementation

Make use of 64-bit registers in 32-bit code

- Use of 64 bit registers while staying link-compatible to existing 32-bit code.
- 64-bit registers are always available when running in z/Architecture mode (64 bit kernel)
- 32 bit code is generally faster due to more efficient cache usage
  - ✓ Useful for small applications not exceeding the 2GB address space
- GCC, Linux Kernel, GNU C library, binutils, and GDB support needed.
  - ✓ Linux kernel with the “highgprs” feature required (see /proc/cpuinfo)
  - ✓ Executables are prevented to be used with older kernel versions.
  - ✓ New core dump format including the upper register halves supported by the kernel and GDB.
- Enabled with compile parameters “m31 mzarch”.
  - ✓ Not enabled by default yet due to problems with inline assemblies, but we are working on this
- Available with SLES11 SP2 add-on GCC and RHEL6.1

GCC s390: 64-bit registers in 31-bit code

```
long long
foo (long long a, long long b,
     long long c)
{
  return a * b / c;
}
```

OLD: -m31
3 x mul + function call!

```
foo:
        stm     %r14,%r15,56(%r15)
        lr      %r1,%r2
        ahi     %r15,-96
        msr     %r4,%r3
        msr     %r1,%r5
        mlr     %r2,%r5
        ar      %r1,%r4
        ar      %r2,%r1
        lm      %r4,%r5,192(%r15)
        brasl   %r14,__divdi3
        lm      %r14,%r15,152(%r15)
        br      %r14
```

NEW: -m31 -mzarch
1 x mul, 1 x div

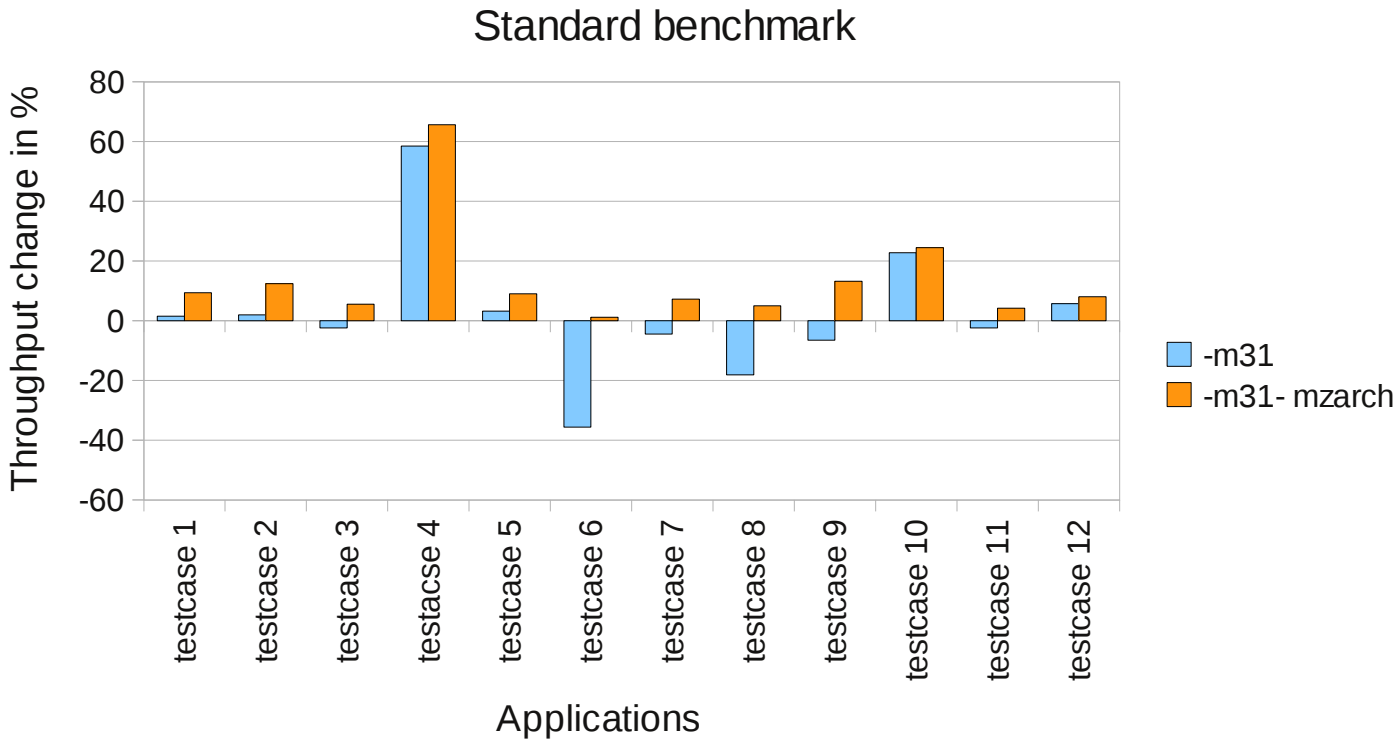
```
foo:
        sllg    %r2,%r2,32
        sllg    %r4,%r4,32
        llgfr   %r3,%r3
        llgfr   %r5,%r5
        ogr     %r3,%r2
        ogr     %r5,%r4
        msgr    %r3,%r5
        dsg     %r2,96(%r15)
        lgr     %r2,%r3
        srlg    %r2,%r2,32
        br      %r14
```

Still the best: -m64

```
foo:
        msgr    %r2,%r3
        lgr     %r1,%r2
        dsgr    %r0,%r4
        lgr     %r2,%r1
        br      %r14
```

GCC s390: 64-bit registers in 31-bit code

- -m64 / -m31 show comparable average performance with tested applications
- -m64 / -m31 -mzarch shows 12,8% improvement
  - ✓ All performance degradations are gone
- Results may vary from application to application



Conclusions and Reference

- Performance experience is influenced by (order by importance)
  - ✓ Application design, programming and proper setup
  - ✓ Hardware capabilities of mainframe models
    - Computation rate, instruction set, cache hierarchy, execution units, number of processors, supported memory, I/O options ...
  - ✓ Highly optimized code tuned to the target machine model
    - Use -O3 or -O2 compile parameter
    - Proper machine tuning with -march / -mtune
    - Link Time Optimization and Feedback Directed Optimization
  - ✓ Use of current GCC versions
    - New internal technologies and features
    - Latest machine and instruction set support
    - In some distributions there is an (unsupported) newer optional compiler included, which has advantages

- More GCC related information is provided on the Linux Performance website:

http://www.ibm.com/developerworks/linux/linux390/perf/tuning_compiler.html

Questions

- Further information is located at
  - ✓ Linux on System z – Tuning hints and tips
    http://www.ibm.com/developerworks/linux/linux390/perf/index.html
  - ✓ Live Virtual Classes for z/VM and Linux
    http://www.vm.ibm.com/education/lvc/

Mario Held

Linux on System z
System Software
Performance Engineer

IBM Deutschland Research & Development
Schoenaicher Strasse 220
71032 Boeblingen, Germany

Phone +49 (0)7031–16–4257
Email mario.held@de.ibm.com