



Vrije Universiteit Brussel

Faculty of Science and Bio-Engineering Sciences
Department of Computer Science

Performance and Efficiency Analysis of Modern Accelerators

Fine-Grained Parallelism on the Intel Xeon Phi

Graduation dissertation submitted in partial fulfilment of the requirements for the degree of
Master of Science in Applied Science and Engineering: Computer Science

Robrecht Dewaele

Promoters: Prof. Dr. Jan Lemeire
Prof. Dr. Roel Wuyts

Adviser: Petar Marendić

JUNE 2014





Performance and Efficiency Analysis of Modern Accelerators

Fine-Grained Parallelism on the Intel Xeon Phi

Proefschrift ingediend met het oog op het behalen van de graad van
Master of Science in de Ingenieurswetenschappen: Computerwetenschappen

Robrecht Dewaele

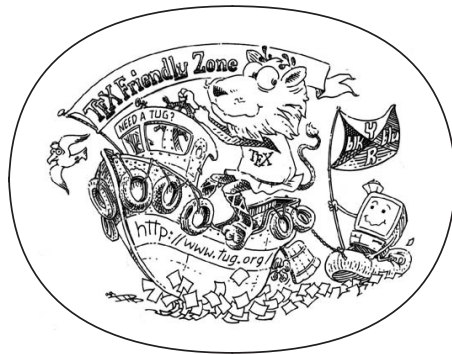
Promotoren: Prof. Dr. Jan Lemeire
Prof. Dr. Roel Wuyts

Begeleider: Petar Marendić



PERFORMANCE AND EFFICIENCY ANALYSIS OF MODERN ACCELERATORS

ROBRECHT DEWAELE



Fine-Grained Parallelism on the Intel Xeon Phi
Bachelor in Computer Science
Department of Computer Science
Faculty of Science and Bio-Engineering Sciences
Vrije Universiteit Brussel

26th June 2014 version 1.1 β

“That day I understood that this heart scares easily. You have to trick it, however big the problem is. Tell your heart, ‘Pal, all is well. All is well.’ ”

“Does that solve the problem?”

“No, but you gain courage to face it.”

— Rancho & Raju (Three Idiots, 2009)

We have seen that computer programming is an art, because it applies accumulated knowledge to the world, because it requires skill and ingenuity, and especially because it produces objects of beauty.

— Donald E. Knuth [Knu74]


ABSTRACT

Supercomputers define the pinnacle of computational power and are an essential tool for solving vast scientific computational problems. They employ increasingly parallel architectures to ever increase their nominal peak performance and to allow them to solve larger problems. Employing the vast amount of computation power is however difficult and optimising for many-core architectures has become of paramount importance. This dissertation aims to obtain an in-depth understanding of modern high-performance accelerators by theoretical study and practical in-depth exploration of the hardware characteristics. The focus lies with the Intel[®] Xeon Phi[™] co-processor, which powers the fastest supercomputer to date. A generic and portable benchmark framework that caters to highly configurable and reusable (micro-) benchmarks has been developed in support of the practical exploration. The effective exploration resulted in the successful identification of new and interesting properties regarding the behaviour of the Xeon Phi[™].


SAMENVATTING


Het absolute toppunt van computationele rekenkracht wordt gedefinieerd door supercomputers. Ze zijn een essentieel onderdeel in de aanpak van immense computationele problemen in de wetenschap. Hun theoretische rekenkracht wordt alsmaar verbeterd door het gebruik van parallele architecturen wat de aanpak van vooralsnog onoplosbare problemen mogelijk maakt. Het is echter niet eenvoudig om de enorme hoeveelheid aan rekenkracht aan te wenden op een doeltreffende manier; optimalisatie voor veelkernarchitecturen is heden van het grootste belang. Deze verhandeling tracht een grondig begrip van moderne, hoogpresterende rekenversnellers naar voren te brengen. Hiertoe wordt zowel een theoretische studie van de architectuur als een praktijkverkenning van zijn realisatie aangewend, wat een afschatting van de prestatiekenmerken mogelijk maakt. De Intel[®] Xeon Phi[™] gekoppelde hulpprocessor staat in voor het leeuwendeel van de rekenkracht geleverd door de vooralsnog krachtigste supercomputer ter wereld en vormt het hoofdonderwerp van deze studie. Ter ondersteuning van de praktische verkenning werd er een generisch en draagbaar raamwerk ontworpen met als doel het ontwerp herbruikbare en verstelbare ijkingprogramma's sterk te vereenvoudigen. De eigenlijke verkenning van de Xeon Phi[™] processor resulteerde in de geslaagde onderscheiding van nieuwe en opmerkelijke eigenschappen betreffende zijn prestatiegedrag.


ACKNOWLEDGEMENTS


 would like to express my sincere gratitude to all the people that supported me throughout the years and without whom this work could not have transpired. I have many friends, colleagues and fellow student to thank, not only for making this work possible, but also for making me realise that I am surrounded by great people with both great hearts and great minds. I was able to stand on the shoulders of giants and would never have been able to be where I am without them.

I can not possibly list my expressions of gratitude in any sensible order. However, I do like structure and am averse to pseudo-randomness so I'll list people according to how long I have known them. It is a mere coincidence that this scheme results in my parents being listed first; it is also a happy coincidental error that my girlfriend is listed second¹. Just one more thing: if you decide to read my acknowledgements, please read them all. I'll tell on you!

irst and foremost I have my parents, Jacques and Marleen, to thank for their care and love they have never ceased to show. I thank them for the opportunities they gave me, and for their continued support, belief and sacrifices through difficult times. I consider myself to be very lucky to have parents that never doubted my capabilities and helped me find my way to what culminates in this dissertation. They provided me with a place I am very proud to call home and I will be forever grateful.


ilith, my love and partner in life, you had to miss me many times when I had to work late. Even this text I am writing in your absence, but I hereby solemnly promise to make up for lost time. You stood always by my side and ensured I took time off work when I needed to, even though I rarely knew that I did myself. You have kept me sane — well ... — and I can not thank you enough, even though I hope to have a life-long opportunity to try. I don't know where I'd be without you, but that doesn't matter because I wouldn't like it anyway.

echt, my dear best friend, I don't know where to begin. You have known me for over fifteen years, and I wouldn't be the person I am without you. We had many good times and shared many laughs but you always gave me the proverbial kick in the pants when I needed it. You'll always find ways to help me. Case in point: I am writing this at your place and it's three in the morning. You don't care, you just want me to succeed in whatever endeavours I embark on, although quietly because you're asleep. Well, you just woke up for a potty break; I think this event will make for a mildly amusing anecdote in ten years time. Cheers!


ries, my other it's-not-a-competition-but-there-can-be-only-one-as-best-is-a-superlative best friend, you are most likely the kindest person I know. It is impossible to count the number of times I benefited from your advice. We haven't been doing lots of the fun stuff together lately, but I get to thank you for proof-reading once more my work. However, it's totally your fault if any errors remain. Hmm ..., I probably shouldn't be trying to be


¹Sarcasm, for should it not have emerged from the silence of the written word.


funny with the lack of sleep I'm having, but this is the acknowledgements section so I get to do what I want!


 an Lemeire, my promoter, I have you to thank for so much more than your work, time and effort for this dissertation. I must thank you for giving me the opportunity to finish my studies, as I honestly believe I couldn't have without your support and kind heart. You provided me with a place to work and showed patience when I needed it the most. I am not sure how to thank you properly, but let it be known that I consider myself to be in your debt and that I am very grateful for what you did.

You have given me the opportunity to work on a dissertation that I take a strong personal interest in and have introduced me to many people that work in the fields I aspire. You left me with a strong degree of freedom and while you were sometimes worried about me, I don't think you ever lost your confidence. I am very glad that you agreed to be the promoter for this thesis.

 an G., office colleague, trusty train companion, and victim of many accidental discussions about my dissertation. As a principal author of the performance model that I employ in this dissertation, you have often provided me with interesting discussions, new insights and observations. You are always helpful and up for a conversation. I thank you for contributing to the very pleasant atmosphere in our office.

 etar Marendić, my adviser and another victim of many discussions on about my dissertation, albeit less accidental. I have disturbed you many times while you were working for which I am sorry, but you never held it against me. Thank you for your insights and advice, not only pertaining to my dissertation, but also to me as a person and as an aspiring computer scientist.

 oel Wuyts, my co-promoter, I am sorry that I haven't know you any longer because it does feel wrong to see your name this low on the list. However, I committed to a scheme and I must stick to it. Roel, I feel privileged to have had you as a co-promoter and you are a great teacher to me. I regret the distance between our offices because I always thoroughly enjoy our conversations. I do hope we can meet again in the future, or in your words: "when all this is done".

 veryone else I have to thank are the people at Infogroep that provided me a place to work in the weekends, a place to host my server and back-ups, and a place I used to call second home; the people at [ETRO](#) that never shied away from a conversation; Adriaan for the inspiration of some of the more mathematical twists in this dissertation; Jan L.¹ for the enlightening discussions on Instruction Set Architectures.

¹Yes, that's three people named Jan, and this one is not my promoter. But do not fret, there are yet more people named Jan at [ETRO](#); it sure is a popular first name.

CONTENTS

1	INTRODUCTION	1
2	THE ROAD TO EXASCALE COMPUTING	5
2.1	General Performance Metrics	5
2.2	Supercomputer Metrics	6
2.2.1	Peak Performance	6
2.2.2	TOP500	6
2.2.3	Gordon Bell Prize	7
2.2.4	Software Application Efficiency	7
2.2.5	Performance Per Watt	8
2.3	Single To Many Core	9
2.3.1	Floating-point Operations Per Cycle	9
2.4	Practical Peak Performance	10
2.4.1	TH-2 & Titan Peak Performance	11
2.5	Current Hypothetical Peak Performance	12
2.6	Near Future	13
3	PROCESSOR DESIGN & INTEL [®] XEON PHI [™]	15
3.1	General Processor Design	15
3.1.1	Instruction Set Architecture (ISA)	15
3.1.2	Dynamic-Static Interface (DSI)	16
3.1.3	Instruction Pipelines	17
3.2	Memory	21
3.2.1	Coherency	21
3.2.2	Optimising Transactions	22
3.3	Many Integrated Core: Intel [®] Xeon Phi [™]	25
3.3.1	Processor Cores	25
3.3.2	Memory Organisation	27
3.3.3	Page Tables	29
3.3.4	Coherency Protocols	30
4	A MODEL FOR FINE-GRAINED PARALLELISM	33
4.1	The Need for a Fine-Grained Performance Model	33
4.1.1	Online vs. Offline Efficiency Analysis	34
4.2	Outline of the Pipeline Model	34
4.2.1	The Pipelines of the Pipeline Model	34
4.2.2	Hardware Performance	35
4.2.3	Software Efficiency	36
4.3	Pipeline Simulations	36
4.4	The Pipeline Model for CPU and MIC Architectures	38
4.4.1	CPU Dissimilarities	38
4.4.2	MIC Dissimilarities	38
4.4.3	A Simple Solution	39
4.5	Related Work	39
5	ADHD: A BENCHMARK FRAMEWORK	41
5.1	Philosophy of the Framework	41
5.2	Range Abstraction	42
5.2.1	Composition	42
5.2.2	Steppers	42
5.3	Benchmarks are Ranges	43
5.4	Multi-Threaded Benchmarks	43
5.4.1	Execution in Phases	44
5.4.2	Synchronised Thread Start	45
5.5	Timing Information	45
5.5.1	Reporting	46

6	BENCHMARK DESIGN	49
6.1	Observing Elapsed Cycles	49
6.1.1	Time Stamp Counter	49
6.1.2	Hardware Performance Counters	50
6.2	Computations	50
6.3	Memory	50
6.3.1	Effective Micro-Benchmark Design	51
6.3.2	Generalisation & Parameterisation	53
6.3.3	Throughput	57
6.3.4	Bandwidth	59
7	BENCHMARK PERFORMANCE & EFFICIENCY ANALYSIS	61
7.1	Xeon [®] E5-2690 v2: Baseline Results	61
7.1.1	Chasing Pointers	61
7.2	Xeon Phi [™] : Detailed Analysis & Contexts	71
7.2.1	Memory Hierarchy Overview	71
7.2.2	Disabled Loop Unrolling & Address Generation Interlock	75
7.2.3	Enabled Loop Unrolling & Register Spilling . .	78
7.2.4	Alignment for L2 Random Access	79
8	CONCLUSIONS	81
8.1	Future Work	82
	BIBLIOGRAPHY	85

LIST OF FIGURES

Figure 2.1	evolution of the TOP500 and Bell Prize performance awards	8
Figure 2.2	power efficiency evolution	8
Figure 3.1	KNC instruction prefetching & decoding ([Cor14])	26
Figure 4.1	the conceptual pipelines of the pipeline model	35
Figure 4.2	computational pipeline: maximal performance	37
Figure 4.3	performance limited by Instruction-Level Parallelism	37
Figure 4.4	performance limited by concurrent execution constraints	38
Figure 5.1	Xeon Phi™ thread synchronisation	46
Figure 6.1	random access pattern with uniform RNG . . .	56
Figure 6.2	random access pattern; symmetrical binomial RNG ($P_n = 0.5$)	57
Figure 6.3	random access pattern; asymmetrical binomial RNG	58
Figure 7.1	Xeon E5-2690 v2: L1 cache latencies	63
Figure 7.2	Xeon E5-2690 v2: L1 cache latencies - no loop unrolling	65
Figure 7.3	Xeon E5-2690 v2: L2 cache latencies	67
Figure 7.4	Xeon E5-2690 v2: L3 cache latencies	68
Figure 7.5	Xeon E5-2690 v2: RAM latencies	70
Figure 7.6	KNC full memory latency overview	73
Figure 7.7	KNC cache hierarchy latency detail	74
Figure 7.8	KNC linear cache accesses	75
Figure 7.9	KNC linear cache accesses	78
Figure 7.10	KNC random L2 cache access: varying alignment	79

LIST OF TABLES

Table 2.1	Processor peak for TH-2 and Titan.	11
Table 2.2	Node peak for TH-2 and Titan.	11
Table 2.3	TH-2 and Titan peak performance.	12
Table 2.4	Hypothetical peak for latest accelerators. . . .	12
Table 2.5	Hypothetical peak for TH-2 and Titan.	12
Table 3.1	the Dynamic-Static Interface (DSI)	17
Table 3.2	scalar pipeline principle	18
Table 3.3	super-pipeline principle ($\Phi = 2$)	19
Table 3.4	super-scalar pipeline principle ($n = 3$)	20
Table 3.5	super-scalar super-pipeline ($\Phi = n = 2$)	21
Table 3.6	KNC Core Pipeline	27
Table 3.7	Xeon Phi™ Cache Hierarchy	28
Table 3.8	KNC Page Tables	30
Table 3.9	MESI States	31
Table 3.10	GOLS ₃ States	31
Table 6.1	Memory bandwidth design & optimisation . .	59

Table 7.1	Xeon [®] E5-2690 v2 L1 cache: chasing pointers configuration	62
Table 7.2	Xeon [®] E5-2690 v2L1 cache: raw data for single thread, no Instruction-Level Parallelism (ILP)	62
Table 7.3	Xeon [®] E5-2690 v2 L2 cache: chasing pointers configuration	66
Table 7.4	Xeon [®] E5-2690 v2 L3 cache: chasing pointers configuration	68
Table 7.5	Xeon [®] E5-2690 v2 RAM: chasing pointers configuration	69
Table 7.6	no unroll, ILP 1: 4 loop cycles	76
Table 7.7	no unroll, ILP 2: 6 loop cycles	76
Table 7.8	no unroll, ILP 14: 26 loop cycles	77

LISTINGS

Listing 3.1	Address Generation Interlock	28
Listing 6.1	Reading the Time Stamp Counter	49
Listing 6.2	Chasing Pointers: Principle	54
Listing 6.3	Chasing Pointers: Setup for Linear Access	54
Listing 6.4	Chasing Pointers: Setup for Random Access	55
Listing 6.5	Chasing Pointers: Exploiting ILP — <i>arrays</i>	57
Listing 6.6	Chasing Pointers: Exploiting ILP — <i>local variables</i>	58
Listing 7.1	ILP 13: no register spilling	62
Listing 7.2	ILP 14: register spilling	64
Listing 7.3	no ILP: no loop unroll	64
Listing 7.4	no ILP: automatic loop unroll	65
Listing 7.5	no unroll: ILP 1	75
Listing 7.6	no unroll: ILP 2	76
Listing 7.7	no unroll: ILP 14	77
Listing 7.8	ILP 13: register spilling	78

DEFINITIONS

1	throughput	5
2	latency	5
3	bandwidth	5
4	Instruction Set Architecture (ISA)	15
5	Dynamic-Static Interface (DSI)	16
6	λ — issue latency	35
7	Λ — completion latency	35
8	concurrent execution constraints	35
9	instruction dependency graph	36

10	instruction context	36
----	-------------------------------	----

ACRONYMS

ACM	Association for Computing Machinery	7
ADHD	Analyze Diverse Hardware Demeanors	83
AGI	Address Generation Interlock	75
AGU	Address Generation Unit	54
ALU	Arithmetic Logic Unit	
APU	Accelerated Processing Unit	13
AVX	Advanced Vector eXtensions	10
CISC	Complex Instruction Set Computer	16
CPI	Cycles Per Instruction	26
CPU	Central Processing Unit	81
CROM	Control Read-Only Memory	
DAG	Directed Acyclic Graph	36
DE	Decode	17
DP	Double-Precision	6
DSI	Dynamic-Static Interface	15
ECC	Error Checking & Correction	25
ETRO	Department of Electronics and Informatics	81
EX	Execute	17
FGP	Fine-Grained Parallelism	33
flop	floating-point operation	6
FMA	Fused Multiply-Add	6
FPU	Floating-Point Unit	22
GOLS ₃	Globally Owned, Locally Shared	
GPGPU	General-Purpose computing on GPU	39
GPR	General-Purpose Register	50
GPU	Graphics Processing Unit	81
HPC	High-Performance Computing	81
HPCG	High Performance Conjugate Gradient	7
HPL	High-Performance Linpack	6
HSA	Heterogeneous Systems Architecture	13
IA ₃₂	Intel Architecture 32-bit	54
IF	Instruction Fetch	17
IL	Issue Latency	36
ILP	Instruction-Level Parallelism	82
IMCI	Initial Many Core Instructions	25
I/O	Input/Output	
IP	Instruction Parallelism	36
IPC	Instructions Per Cycle	25
ISA	Instruction Set Architecture	15
ISC	International Supercomputing Conference	9
KNC	Knights Corner	81
KNL	Knights Landing	83
LRU	Least Recently Used	28
MESI	Modified Exclusive Shared Invalid	
MIC	Many Integrated Core	81
MMU	Memory Management Unit	29
MMX	Multi-Media eXtensions	10
MOESI	Modified Owned Exclusive Shared Invalid	

MP	Machine Parallelism	44
OL	Operation Latency	50
OOE	Out-of-Order Execution	83
OpenCL	Open Computing Language	83
OS	Operating System	29
PAPI	Performance Application Programming Interface	45
PCIe	Peripheral Component Interconnect Express	25
RAM	Random-Access Memory	69
RAW	Read After Write	51
RDTSC	Read Time Stamp Counter	49
RISC	Reduced Instruction Set Computer	16
RNG	Random Number Generator	56
SCAT	Symmetric Cores and Asymmetric Threads	39
SIMD	Single Instruction Multiple Data	10
SM	Streaming Multiprocessor	10
SMP	Symmetric Multi-Processing	40
SP	Single-Precision	6
SSE	Streaming SIMD Extensions	23
TD	Tag Directory	30
TH-2	Tianhe-2	81
TLB	Translation Lookaside Buffer	71
TLP	Thread-Level Parallelism	33
TSC	Time Stamp Counter	49
USWC	Uncacheable Write Combining	23
VUB	Vrije Universiteit Brussel	81
WAR	Write After Read	51
WAW	Write After Write	51
WB	Write Back	18

INTRODUCTION

This thesis aims to provide a qualitative and in-depth understanding of modern High-Performance Computing (HPC) hardware. Such knowledge is most often required for creating highly efficient software. The approach is chiefly based on three pillars: hardware characterisation by micro-benchmark analysis, application characterisation by source code analysis, and a performance model to tie both characterisations together.

This dissertation focuses on the Intel® Xeon Phi™ and its Many Integrated Core (MIC) architecture as principal use-case. It is a relatively new HPC co-processor, but already powers the world's fastest supercomputer (Tianhe-2) to date: the Intel® Xeon Phi™. The performance model of choice is the result of ongoing internal research at Department of Electronics and Informatics (ETRO) and is currently aimed towards Graphics Processing Units (GPUs). This dissertation presents a minor reinterpretation of the model to apply it on the Xeon Phi™. Simplified versions of the tests are performed on a Xeon® Central Processing Units (CPUs) to provide a frame of reference. The process of running a large number of experiments is greatly facilitated by a purpose-built and portable benchmark framework.

The remaining sections describe each chapter in more detail and provide a reference frame for the work presented therein.

PERFORMANCE & EFFICIENCY IN SUPERCOMPUTING

Sequential software performance has been improving at an exponential rate as a direct consequence of Moore's law [Moo98]. This rate has been slowing down however: it averaged approximately 55 % per year before 2003, but has decreased to approximately 22 % per year since [HP11]. This decline is mainly due to a physical limitation known as the CPU frequency power wall: power consumption grows exponentially in function of operating frequency. The main technique to alleviate this problem has been the move to multi-core systems, where a single chip comprises multiple cores running at a lower frequency. Both performance potential and power consumption increase linearly with the number of cores. This does however imply that the free lunch for software is over and optimising for many-core devices has become of paramount importance.

The trend of supercomputer performance improvements since 1988 suggests machines will be capable of executing over 10^{18} calculations per second in 2018. In practice however, a significant amount of performance potential is wasted by software as running software on many-core systems incurs a high overhead. For example, the highly optimised High-Performance Linpack (HPL) [Pet+12] benchmark only succeeds to employ 66 % of the world's fastest supercomputer to date, Tianhe-2 (TH-2). Astonishingly, it wastes more absolute compute power than obtained by the most recent winner of the Gordon Bell prize, which awards exceptional achievements in HPC yearly. Not only potential advancements in science are hampered; The scale of these

A means to facilitate writing highly efficient software for modern fine-grained processor architectures.

*Chapter 2;
Software efficiency is key to unlock the exascale performance levels hardware will have to offer by 2018.*

machines incurs a high financial cost through maintenance and power consumption as well.

An increase in software efficiency results directly in increased performance for genuine applications. While this is an interesting result by itself, it also improves the practical power efficiency of the machine by wasting less cycles. Shorter running times enable tackling larger problems, or freeing up time to run other applications. Better power efficiency lowers the cost of operating a supercomputer and increases its viable lifetime. On the small scale, increasing software efficiency may make it feasible to run an application on a home computer instead of a central server, thereby increasing the comfort and mobility of the end user.

PROCESSOR DESIGN

*Chapter 3;
Maximising
software efficiency
requires a good
understanding of
processor technology.*

As illustrated by Chapter 2, theoretical peak performance is nigh impossible to obtain by proper software applications. Genuine applications incur many kinds of overheads that result in reduced performance. A qualitative understanding of the target execution architecture is required to minimise or eliminate computations that do not functionally contribute to the outcome of the program. Additionally, specific hardware features may be exploited to further increase efficiency and performance.

The main subject for the practical studies conducted in this thesis is the Xeon Phi™ co-processor. It is based on Intel®'s MIC architecture and processor architecture discussions will mainly focus thereupon. A basic study on general CPU architectures provides a reference to the Xeon Phi™ core design and highlights important differences.

MODELLING SOFTWARE EFFICIENCY

*Chapter 4;
The GPU pipeline
model facilitates
understanding
software efficiency
through analysis of
lost cycles and may
be reinterpreted to
model the MIC core
architecture.*

Theoretical hardware performance can easily be inferred from its technical specifications as shown in Chapter 2. However, reaching this limit in practice for non-trivial programs proves to be a complex endeavour due to software overheads. These inefficiencies range from avoidable additional computations to unused processor cycles, i.e. *lost cycles*. Manual analysis and profiling is a meticulous and time-consuming task which can be alleviated through the use of performance models.

The Pipeline Model

The pipeline model is the result of ongoing research at Vrije Universiteit Brussel (VUB), ETRO and was inspired by GPU architectures. Contemporary GPU designs focus on large-scale exploitation of Fine-Grained Parallelism (FGP): instructions from different contexts can easily be executed back-to-back without significant overhead. The pipeline model uses effective hardware instruction latencies and software execution contexts to predict application efficiency. It supports a priori analysis to determine the most suitable hardware architecture for a given application, and a posteriori analysis to precisely locate possible efficiency problems.

The practical instruction latencies are not expected to deviate significantly from the processor hardware specifications. However, when software performance is predominantly limited by a hardware sub-

system, the hardware will have to delay execution, thereby increasing practical instruction latency. Such behaviour can cause significant variations in instruction latencies which will compromise the model's accuracy. By itself, this is a property of the application software, but the *impact* of such inefficiencies is a property of the hardware. The pipeline model employs *contexts* to take this effect into account. Contexts allow different latencies to be used for different parts of the application software.

The measurement of practical latencies and the impact of software inefficiencies can be performed using micro-benchmarks. These are extremely small programs to stress the hardware and approach theoretical performance limits in practice. They can also be used to explore the hardware for potential significant bottlenecks which then can be modelled as contexts. While measuring latencies is systematic process, the search for contexts is an experimental process and requires creativity and educated exploration.

BENCHMARK FRAMEWORK

All benchmarks are implemented in a benchmark framework, developed specifically for this thesis. It is written in portable C++11, that in combination with careful design and implementation of concrete benchmarks make it possible to re-use the software on any platform supported by a C++11 compiler. Its versatility is further illustrated by running all benchmarks on CPUs as well, whose results serve as a reference point for the results obtained on the Xeon Phi™ co-processor. High configurability facilitates fine-tuning the benchmarks to different platforms. It also enables measuring the impact of sub-optimal configurations, which illustrates the impact certain components have on performance.

Micro-benchmarks are often written directly in machine code as opposed to a higher level language to avoid complexities introduced by a toolchain, such as compiler optimisations. In contrast, this thesis proposes a set of micro-benchmarks written in an application level programming language, as they yield much more relevant information: besides being tools to measure practical peak performance characteristics, they also illustrate how to express code in said programming language to obtain these results.

MICRO-BENCHMARK DESIGN

Initial development and testing of the micro-benchmark is possible without a Xeon Phi™ considering its similarities with CPU architectures. A set of essential instruction latency micro-benchmarks can be devised based on hardware specifications alone. The identification of pipeline model contexts is however an empirical search process which requires a basic amount of creativity and inventiveness.

A discussion the results of executing these benchmarks is left to the following chapter, as the next section* will detail.

Chapter 5;
A benchmark framework facilitates the development a large amount of generic, reusable micro-benchmarks.

Chapter 6;
The design of efficient, relevant and accurate micro-benchmarks through application of a qualitative understanding of the MIC architecture.

*Chapter 7;
Micro-benchmarks
do not only measure
software efficiency,
but also illustrate
the implementation
strategy for optimal
performance in a
high-level language.*

BENCHMARK RESULTS

The exploration of the performance characteristics of the Xeon Phi™ takes a systematic approach to search for and identify anomalies in performance. More detailed and focused micro-benchmarks are set up by interpreting the results of earlier and more generic ones.

Surprisingly, the benchmark results will be able to identify a small but nevertheless significant inaccuracy in how the Xeon Phi™ programming manual details the instruction scheduling of the core. It will additionally show that the Intel® compiler may generate inefficient code for memory transactions when certain conditions are met.

The benchmarks will also identify a number of contexts and find necessary latencies for the pipeline model. A summary is presented in the conclusive Chapter 8.

CONCLUSION

*Chapter 8;
Developing highly
efficient software for
architectures using
fine-grained
parallelism now has
a more systematic,
re-usable and
future-proof
approach.*

The last chapter summarises the main findings of this thesis:

- We were successfully able to apply the pipeline model to the Xeon Phi™ core and we identified multiple latencies and contexts.
- The Analyze Diverse Hardware Demeanors (ADHD) benchmark framework proved to be a great asset in the general analysis of the Xeon Phi™ behaviour.
- We were able to obtain the hardware-specified latencies for first-level cache, which is a, effective improvement on the state of the art [Fan+14]. Additionally we have shown that the second-level cache performance may vary significantly and identified two main causes.
- We identified minor but profoundly confusing inaccuracies in the Xeon Phi™ programming guide and clarified how the core instruction scheduling mechanisms operate.
- We have shown that the Intel® compiler generates sub-optimal code in certain circumstances.

This manuscript finally concludes with a short list of possible interesting future work, such as the analysis of the Xeon Phi™ successor, to be released in 2015 or the extension of the framework to GPU accelerators.

This chapter presents the Intel® Xeon Phi™ co-processor and its position in the world of High-Performance Computing (HPC). Future Xeon Phi™ generations will be instrumental to the performance of the supercomputers of tomorrow. Today, it provides the main processing power of the fastest supercomputer to date, the Tianhe-2, and plays an important role in Intel's commitment to achieve *exascale* computing [Skat11] by 2018.

The current trends in supercomputer performance suggest that theoretical peak performance will likely achieve an exascale level by 2018, but that software efficiency limitations may cause performance to lag behind by more than a year. If the exponential growth in computer performance continues, software will enter the exascale era at 50 % efficiency in 2019.

Obtaining quasi-peak performance is extremely hard; the highly parallel architectures of supercomputers and their components require problems that the available parallelisms in problem specifications are almost trivially derivable. This chapter introduces different metrics for supercomputer performance, their importance and their relevance. It motivates the need for improvements in software efficiency and shows that much headroom remains.

2.1 GENERAL PERFORMANCE METRICS

Performance can essentially be conceptualised as *throughput*, i. e. the ratio of completed work to unit time. *Latency* is a property inherent to a basic unit of work, i. e. the time needed to complete a single task. When only one basic work of unit is performed at a time, throughput is uniquely determined by latency. In practice, many methods are used to increase throughput by overlapping multiple requests and thereby *hiding* their inherent latencies, e. g. instruction pipelines, parallel execution, speculative execution, et cetera (Chapter 3). By measuring both properties in isolation, micro-benchmarks may estimate the impact of such techniques on practical performance (Chapter 6). A third measure of performance is *bandwidth*, which is synonymous to throughput, but preferred terminology depending on the context. In their most generic form, the three performance concepts can be defined as follows:

Definition 1 (throughput)

Throughput is defined as the number of satisfied requests over time. It is the rate at which requests can be satisfied. Throughput is typically expressed in terms of unit time or compute cycles.

Definition 2 (latency)

Latency is defined as the chronological distance between issuing a request and receiving a response, expressed in compute cycles.

Exa-scale computing will be available by the year 2018 if computing power continues to improve like it has for the past twenty-five years.

Definition 3 (bandwidth)

Bandwidth is essentially equivalent to throughput, but is used in different contexts. Bandwidth is typically expressed in terms of processed data over time.

2.2 SUPERCOMPUTER METRICS

Supercomputers are designed to solve extremely large computational problems present in science. Such workloads make heavy use of real numbers (\mathbb{R}) which potentially require infinite precision. Computers are finite machines and use a standard 32 bit Single-Precision (SP) or 64 bit Double-Precision (DP) *floating-point* format to represent such values. This makes the rate at which a computer can execute operations on floating-point values — expressed as floating-point operations (flops) per second — of critical importance. The rates will typically be different for SP and DP workloads, due to their different sizes. In practice, double-precision performance is preferred as its increased precision is often a must in scientific workloads and therefore most relevant.

It is important to note that a flop does not necessarily equate to a floating-point *instruction*. Specialised instructions model often-used operation compositions as a single instruction in order to increase flop rate potential. The Fused Multiply-Add (FMA) instruction for example will perform a multiplication and an addition, i. e. 2 flop, and has the added benefit that rounding of the result will occur only once.

2.2.1 Peak Performance

Theoretical peak performance can be calculated from hardware specifications, but does not account for possible slowdowns caused by sub-components or software inefficiencies. Benchmarks estimate real-world performance more accurately, but can not represent actual application performance as their scope is limited in order to remain adaptable to different systems. Application-specific performance will be limited by some specific sub-component(s) of a machine, but evidently has the definitive answer to which hardware handles it better. Which process is best to obtain relevant flop rates will depend on the specific comparison that is being made.

2.2.2 TOP500

The TOP500 project [Str+] ranks and details the most powerful single-site (i. e. not geographically distributed) computer systems in the world as determined by the High-Performance Linpack (HPL) benchmark [Pet+12]. In 2002 it added total power requirements to its statistics in accordance to the growing importance of power efficiency. The fastest supercomputer as of 2013 is the Tianhe-2 (TH-2) machine, located in China. It is able to run HPL at 33.86 Pflop/s and has a theoretical peak performance of 54.9 Pflop/s (also see Section 2.4.1).

HPL measures floating-point computing power by solving dense systems of linear equations. While it has been the main metric for several decades to measure supercomputer performance, its relevance to current HPC and genuine HPC applications is waning. The benchmark is generally considered to be increasingly unreliable as a true measure of system performance for a growing collection of important science

and engineering applications. A likely successor to HPL is the High Performance Conjugate Gradient (HPCG) benchmark [DH13], which focuses on computations and data access patterns more commonly found in modern genuine HPC applications.

2.2.3 Gordon Bell Prize

While relevant and relatively adequate to estimate potential computing power, HPL performance figures are not fully representative of genuine applications. The history of *Gordon Bell Prize* [Com] provides us with a better means to this end. The Association for Computing Machinery (ACM) awards the prize each year, as a recognition for outstanding achievements in high-performance computing. Winning applications broke the giga-, peta- and teraflop/s barriers in respectively 1988, 1999 and 2009. These breakthroughs are detailed in their corresponding award-winning papers, [Bro+89; Mir+99; Eis+09].

The winners in 2013 reached 55 % efficiency (10.99 Pflop/s) simulating cloud cavitation collapses on the *Sequoia* supercomputer [Ros+13]. Sequoia has a theoretical peak performance of 20.132 Pflop/s and is currently ranked third on the TOP500 list with 17.137 Pflop/s. This illustrates the optimism displayed by HPL, as it is able to run with 85 % efficiency on Sequoia.

2.2.4 Software Application Efficiency

In a 2001 report, Gao [Gao01] proposed that achievable performance will not be able to keep up with the advances in microprocessor technologies. The report includes a plot of both hardware and software performance evolution to support this argument. As a continuation, figure 2.1 shows the evolution of TOP500 number-one machines in terms of theoretical peak performance (R_{peak}) and maximal HPL performance (R_{max}). R_{lost} plots the absolute difference between R_{max} and R_{peak} , i. e. *lost* performance. Additionally, it shows the peak performance obtained by winners of the Gordon Bell prize. This provides an *indication* of genuine application performance evolution; Award-winning performance is not always obtained on the fastest supercomputer available.

HPL performance continually improves by definition, but its efficiency fluctuates as new supercomputers take the top spot. On the TH-2 in particular, it is only able to employ 62 % of the available compute power. Genuine application performance does not seem to be able to keep up with advancements in hardware, as it is dropping further behind both theoretical peak and HPL performance. In 2007 and 2013, Bell prize winners obtained even less flop/s than HPL left unused on the TOP500 list's number one at the time. Designing efficient algorithms that scale well to high-performance computers remains of utmost importance in order to be able to exploit the power next-generation supercomputers will have to offer. This thesis has a strong emphasis on software efficiency and proposes means to easily analyse and assess the impact of decisions in software on real-world performance.

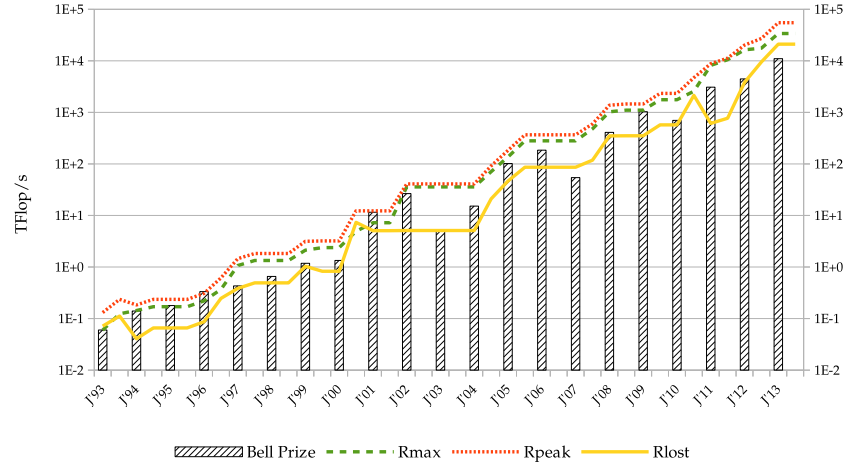


Figure 2.1: evolution of the TOP500 and Bell Prize performance awards

2.2.5 Performance Per Watt

Power requirements induce an increased cost of operation and pose additional engineering challenges in the race to exascale computing. Power efficiency in HPC became increasingly important in the last decade and is now a relevant and important metric.

The Green500 list [Fen+] project started in November 2007 and ranks supercomputers according to their power efficiency, typically expressed in Mflop/W. The recent focus on power-efficient computing has led to exponential improvements during the list's current lifetime. Figure 2.2 shows the power efficiency evolution of both the Green500 and TOP500 lists top machines, and the Green500 median as a baseline. The most power-efficient machine in 2013 was capable of executing 4503.17 Mflop/W, while the fastest machine had an efficiency of 1901.54 Mflop/W.

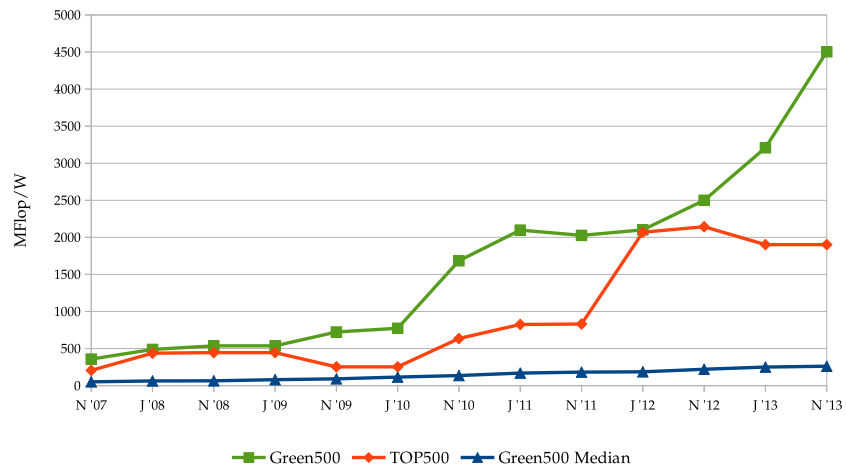


Figure 2.2: power efficiency evolution

Even though at a lower rate, the TOP500 supercomputers have also made major improvements in power efficiency. It is nevertheless improbable that the exponential improvements as shown by the Green500 machines will continue. Cutting-edge semiconductor manufacturing technology uses a 14 nm lithographic process, and further power density improvements become very challenging [War11]. The Green500 staff estimated the power efficiency of an exascale supercomputer in 2018 to be between 67 Mflop/W and 100 Mflop/W in their presentation at the International Supercomputing Conference (ISC) in 2012 [SSF12].

This thesis does not aim to contribute directly in this field. However, increasing software efficiency has the secondary effect of improving practical power efficiency for genuine applications.

2.3 SINGLE TO MANY CORE

The Xeon Phi™ is an important step towards Intel®'s goal building a supercomputer breaking the 1000 Pflop/s (i. e. 1 Eflop/s) barrier. The first supercomputers to break the 1 Gflop/s, 1 Tflop/s and 1 Pflop/s barriers were built in 1988 (*Cray Y-MP*®), 1997 (*ASCI Red*) and 2008 (*Roadrunner*) respectively (also see figure 2.1). It becomes evident why Intel® aims at 2018 to reach exascale computing when extrapolating the trend over the past 26 years.

2.3.1 Floating-point Operations Per Cycle

A processor cycle corresponds to the most basic unit of work a processor can perform. Theoretical peak performance per time unit is therefore the result of a simple multiplication of a processor's clock rate and work performed per cycle (equation 2.1). Clock rates can be obtained from the hardware specifications and can often even be queried from software. Conversely, calculating peak performance per cycle requires basic insight in the micro-architecture of the processor.

$$\text{flop/s} = \text{clock} * \frac{\text{flop}}{\text{cycle}} \quad (2.1)$$

Operations on floating-point values are executed by a Floating-Point Unit (FPU), which is a logical¹ component of a processor core. When processors have multiple FPUs, their peak performance equates to the FPU peak performance multiplied by the total number of units available (equation 2.2). This chapter is solely concerned with peak performance calculations; Chapter 3 introduces processor architectures and details multiple techniques they employ to improve performance.

$$\frac{\text{flop}}{\text{cycle}} = \text{fpu} * \frac{\text{flop}}{\text{fpu cycle}} \quad (2.2)$$

2.3.1.1 CPU

The complete set of instructions a processor can execute is determined by its Instruction Set Architecture (ISA). The now ubiquitous x86

¹In the eighties, it was common for a FPU to also be physically separate from the main processor and to act as a co-processor.

ISA was originally developed by Intel® in 1978 and has been evolving ever since.

Modern processor implementations include many additions to the original x86 instruction set in order to increase the number of *flop* executed *per cycle*. They introduce Single Instruction Multiple Data (SIMD) support with e. g. Multi-Media eXtensions (MMX) or Streaming SIMD Extensions (SSE). The Intel® Ivy Bridge and AMD® Bulldozer architectures also support Advanced Vector eXtensions (AVX), which include SIMD FMA instructions that operate on vectors of up to 256 bit wide, e. g. 8 SP or 4 DP floating-point values. The Knights Corner (KNC) co-processor supports Initial Many Core Instructions (IMCI) which include FMA SIMD instructions for even wider 512 bit vectors.

The vector and fused-operation capabilities are imperative to theoretical peak performance and are represented as *flop per instruction*. A FPU may require multiple cycles to complete an instruction, even in a best-case scenario. In general, both the optimal rate of completed instructions and the amount of *flops* per instruction have to be taken into account, as shown in equation 2.3.

$$\frac{\text{flop}}{\text{fpu cycle}} = \frac{\text{flop}}{\text{instruction}} * \frac{\text{instruction}}{\text{fpu cycle}} * \frac{\text{fields}}{\text{vector}} \quad (2.3)$$

2.3.1.2 GPU

The micro-architecture of the NVIDIA® Kepler Graphics Processing Unit (GPU) is drastically different from the other processors presented in this section and bears no resemblance to x86 implementations. Its basic component is a Streaming Multiprocessor (SM), of which a multitude compose the full GPU. Kepler introduces the next-generation SMX SM architecture. One SMX unit incorporates 192 SP and 64 DP FPUs, all able to complete a scalar FMA instruction every cycle. An SMX is therefore capable of performing 128 flop/cycle for DP data.

2.4 PRACTICAL PEAK PERFORMANCE

The focus of this thesis lies with software efficiency, i. e. the ratio of actual and theoretical peak performance. Real-world performance is a property of software and its execution time, while theoretical peak performance is strictly a hardware property. This section illustrates how peak performance can be calculated from hardware specifications, and applies the formulae to the two fastest supercomputers today. The most prominent difference between the machines is their choice of co-processor architecture: Many Integrated Core (MIC) for TH-2 and GPU for Titan.

TH-2 comprises 16 000 nodes communicating through a custom interconnection network. Each node features 2 Intel® Xeon® E5-2692 processors and 3 Intel® Xeon Phi™ 31S1P co-processors [Don13]. The respective code-names for their micro-architectures are *Ivy Bridge* and *KNC*.

Titan is a Cray® XK7 system with 18688 nodes. Each node contains a AMD® Opteron™ 6274 Central Processing Unit (CPU) and an NVIDIA® Tesla® K20X GPU [Lab12; Wel13]. The micro-architectures for these processors are named *Bulldozer* and *Kepler* respectively.

2.4.1 TH-2 & Titan Peak Performance

All micro-architectures in the scope of this section have a FPU design capable of completing an instruction every clock cycle. This property simplifies the theoretical peak flop/s calculation as the processor clock cycle equals the amount of instructions completed per second (equation 2.4). This is also the formula used in table 2.1 to calculate total peak performance.

$$\text{flop/s} = \text{clock} * \text{fpu} * \frac{\text{flop}}{\text{fpu cycle}} * \frac{\text{fields}}{\text{vector}} \quad (2.4)$$

Ivy Bridge processors have a floating-point unit for each logical core, but the Bulldozer architecture shares a unit between two logical cores. That is to say, the TH-2 12-core CPUs each have 12 FPUs and the Titan 16-core CPUs each have 8. The Xeon Phi™ co-processor of TH-2 has a FPU for each core, i. e. 57 FPUs in total. Further specifications are shown in table 2.1, along with the resulting theoretical peak flop/s.

Processor	FPUs	flop rate ^a	Clock ^b	Peak ^c
Xeon® E5-2692	12	8	2.2	211.200
Xeon Phi™ 31S1P	57	16	1.1	1003.200
Opteron™ 6274	8 (16 / 2)	8	2.2	140.800
Tesla® K20X	896 (14 * 64)	2	0.732	1311.744

^a flop/(fpu cycle) ^b GHz ^c Gflop/s

Table 2.1: Processor peak for TH-2 and Titan.

Compute node performance is shown in table 2.2. Peak performance of the complete machine is the accumulated peak performance of all its nodes. Final theoretical peak performance of TH-2 and Titan is shown in table 2.3.

Machine	CPUs ^a	co-processors ^a	Node Peak ^a
TH-2	2 * 211.200	3 * 1003.200	3432.000
Titan	1 * 140.800	1 * 1311.744	1452.544

^a Gflop/s

Table 2.2: Node peak for TH-2 and Titan.

Despite the 2688 extra nodes in Titan, it has approximately half the computer power of TH-2 because of their significantly lower node performance. While this may seem extraordinary, this is to be expected. Computer performance continues to have an exponential growth (Figure 2.1) and Titan led the TOP500 list the year before TH-2 did.

Machine	Nodes	Node Peak Tflop/s	Total Peak Pflop/s
TH-2	16 000	3432.000	54.912
Titan	18 688	1452.544	27.145

Table 2.3: TH-2 and Titan peak performance.

2.5 CURRENT HYPOTHETICAL PEAK PERFORMANCE

Peak performance of TH-2 seems to be lagging a bit behind the projected 64 Pflop/s for 2014. However, faster models for the Xeon Phi™ and Tesla® co-processors are already available. The hypothetical peak performance of TH-2 and Titan using the latest models of their respective accelerators sketches the most recent evolution of modern HPC hardware. Table 2.4 shows the specifications and the theoretical peak performance of the fastest KNC- and Kepler-based hardware available at the time of writing.

Processor	Cores	flop rate ^a	Clock ^b	Peak ^c
Xeon Phi™ 7120D	61	16	1.238	1208.288
Tesla® K40	960 (15 * 64)	2	0.745	1430.400

^a flop/(fpu cycle) ^b GHz ^c Gflop/s

Table 2.4: Hypothetical peak for latest accelerators.

Table 2.5 shows theoretical peak performance that would result from upgrading the current co-processors of TH-2 and Titan.

Machine	Nodes	CPU's ^a	co-processors ^a	Peak ^b
TH-2	16 000	2 * 211.200	3 * 1208.288	64.756
Titan	18 688	1 * 140.800	1 * 1430.400	29.363

^a Gflop/s ^b Pflop/s

Table 2.5: Hypothetical peak for TH-2 and Titan.

Titan has 2688 more nodes than TH-2, but only one co-processor per node instead of three. The relative performance increase is also comparatively lower, at 9.05 %. However, a deployment of a the Tesla® K40 GPU on the same scale as TH-2 would result in a faster machine yet. Thus far it seems improvements in computer processor technologies will be able to live up to the expectations regarding theoretical peak performance.

2.6 NEAR FUTURE

INTEL[®] is developing the successor to KNC, code-named Knights Landing (KNL). The new co-processor will feature up to 72 cores based on the *Airmont* architecture, and improved inter-core connectivity by switching to a meshed interconnection network [HD]. Airmont will also be used for the Atom[™] processor line, Intel[®]'s power-efficient CPU product line. These changes amount to a radical different design compared to the current Xeon Phi[™] generation.

NVIDIA[®] is working on the successor for the Kepler architecture, code-named *Maxwell* [NVI14]. A major design goal for Maxwell was power efficiency, and the initial focus for the architecture lies with battery-powered devices. There are currently no models available featuring DP units, and are as such not suitable for the HPC market.

AMD[®] does not target the very high-end of the supercomputer market with their current product lineup, where it is overshadowed by NVIDIA[®] as a consequence. The company does offer competitive personal HPC GPU products as part of their FirePro[™] product line. The only machine with AMD[®] GPUs in the top 100 of the Green500 list in November 2013 has a power efficiency of 2351.10 Mflop/W, earning it the eleventh place. It is also present in the TOP500 on position 59, with a HPL R_{\max} and R_{peak} ratings of 532.6 Tflop/s respectively 1098 Tflop/s, which equates to an efficiency of 48.51 %.

Additionally, the company continues to develop their Accelerated Processing Unit (APU) products, which integrate both a CPU and a GPU on a single chip. They follow the principles of Heterogeneous Systems Architecture (HSA), which allows programs to be agnostic to the specific hardware they are running on. The main benefit for HPC is that APUs are very power-efficient.

Understanding the world's fastest hardware for scientific computing is key to unlocking its full potential.

This chapter introduces the Intel® Xeon Phi™ micro-architecture. It provides the essential knowledge needed to construct relevant and precise micro-benchmarks (Chapters 5 and 6) and analyse their results in detail (Chapter 7).

Section 3.1 introduces a set of essential and common concepts in microprocessor design: modern high-performance processors employ Fine-Grained Parallelism (FGP), i. e. parallel execution at the instruction level, as the most important of multiple methods to finish more work in less clock cycles.

Section 3.2 presents a summary of optimisations in the memory hierarchy of computer systems. Entire books may be written on this topic; this section purposefully limits its scope to the effect different components and optimisations in memory systems may have on practical performance.

Section 3.3 lists the specifications of the Intel® Xeon Phi™ MIC implementation code-named Knights Corner (KNC). These sections provide the hardware specifications and allow to assess the efficiency of the micro-benchmarks in Chapter 7.

3.1 GENERAL PROCESSOR DESIGN

The previous chapter introduced a small variety of High-Performance Computing (HPC) processor designs. These are all *instruction set processors*, a highly common type of microprocessors. A fundamental part of instruction set processors is their Instruction Set Architecture (ISA), which plays a crucial role in a processors' design as a whole. It does not only define the set of instructions that the hardware may execute; it also determines the Dynamic-Static Interface (DSI) [MP87] which defines the boundary between interpretation and translation of a program.

The techniques that may be used for efficient instruction processing form the bulk of this section as they ultimately determine the potential performance of the microprocessor. All techniques apply to the *instruction pipeline* and use one of two approaches: they either increase the *efficiency* of the hardware, i. e. improve performance for little or increase in hardware components, or increase the *amount* of hardware, i. e. duplicate different hardware components, or both.

3.1.1 Instruction Set Architecture (ISA)

The set of commands that can be executed by an instruction set processor is defined by its ISA:

Definition 4 (Instruction Set Architecture (ISA))

The Instruction Set Architecture specifies a set of instructions that characterises the functional behaviour of an instruction set processor. It is the interface hardware exposes to software.

An *ISA* may have different implementations, which greatly helps software portability. Backwards compatibility can be ensured by adhering to the same *ISA* as hardware implementations evolve. For example, most (if not all) Intel® and AMD® processors for the desktop market segment support the x86 *ISA* since it was first introduced in 1978. Many approaches exist to *ISA* design, but most are commonly referred to as one of two archetypes:

- Reduced Instruction Set Computer (*RISC*) machines feature a relatively small instruction set. This greatly simplifies their hardware implementation and helps to achieve faster machines.
- Complex Instruction Set Computer (*CISC*) machines include specialised instructions in their instruction set that essentially compose a number of simpler and basic instructions. This increases flexibility in how the machine may execute a program, which generates additional opportunities for optimisation. While these features complicate the hardware implementation, they help to achieve highly efficient execution of software programs.

The *ISA* also determines the so-called Dynamic-Static Interface, as discussed in the following section.

3.1.2 *Dynamic-Static Interface (DSI)*

Software applications are essentially problem specifications, and solving a problem typically involves two stages:

1. *Translation* changes the representation or form of the problem specification.
2. *Interpretation* executes a problem specification and generates results. The execution may require input data that is not part of the specification.

These two stages give rise to the *DSI*, defined as follows:

Definition 5 (Dynamic-Static Interface (DSI))

The Dynamic-Static Interface defines the boundary between the translation and the interpretation stages involved in problem solution.

This distinction however requires a clear definition of what constitutes software. The original authors specified that if an interface can not be changed while the computer is operating, it is part of the hardware. Everything *below* such an interface, i. e. the interfaces it relies on, is also part of the hardware. All other interfaces are part of the software. There is no reason to deviate from the original definition in this dissertation.

Table 3.1 illustrates the *DSI* and how the *ISA* has an impact on its placement. A hypothetical architectures that supports the direct execution of a high-level programming language would place the *DSI* at the highest level; no translation of the software (i. e. compilation) is needed (nor beneficial) prior to execution.

Conversely, *RISC* machines place the *DSI* much closer to the hardware. They require that high-level language code is translated and heavily optimised *before* it is executed.

program ↓	software ↓	compiler complexity	exposed to software	<i>static</i>
architecture				DSI
↓	↓	hardware complexity	hidden in hardware	<i>dynamic</i>
machine	hardware			

Table 3.1: the Dynamic-Static Interface (DSI)

The microprocessors in the scope of this dissertation are CISC machines. They heavily optimise code execution in hardware, and software may be expressed in terms of a relatively rich instruction set. Compilation is still necessary and static optimisation remains highly beneficial.

The following sections will discuss optimisation techniques for increasing the instruction throughput in modern processors, regardless of which archetype they associate.

3.1.3 Instruction Pipelines

Pipelining is an effective technique to increase the throughput of a system with negligible increase in hardware. Its principles are based on the observation that the complexity of a task ultimately determines the minimal time that is required to process it.

In order to reduce the complexity of a task, it may be divided in a sequence of multiple sub-tasks. The complexity of the composite task remains the same, but the complexity of a sub-task is per definition lower and the minimal time needed to complete any single task is reduced. When the same composite task has to be executed multiple times (or different composite tasks share common sub-tasks), the machine may subsequently execute the same basic task for subsequent instructions. This technique is called pipelining. It does not reduce the time required to complete a composite task, but it reduces the minimal delay before being able to start working on the next task. This has the effect that different sub-tasks of different composite tasks are executed at the same time, at a rate only limited by the reduced complexity of a sub-task.

Norm Jouppi presented a classification for pipelined machines according to four parameters, and uses the same amount of conceptual instruction stages to illustrate their design [JW89]. He first defines a scalar baseline machine and compares the different speedups acquired by different instruction pipeline implementations. This section follows the same structure as the original paper but only focuses on the material relevant to dissertation.

The conceptual instruction stages are:

1. Instruction Fetch (IF): Acquire the *opcode*, i. e. the raw instruction data.
2. Decode (DE): Decode the opcode, i. e. extract the operation and its operands.
3. Execute (EX): Execute the operation and generate a result.

4. Write Back (WB): Store the result to memory.

The four parameters are:

- Operation Latency (OL): The number of processor cycles before the result of an instruction may be consumed by a subsequent instruction.
- Machine Parallelism (MP): The maximal amount of instructions that a processor may execute simultaneously.
- Issue Latency (IL): The number of processor cycles required between starting work on two consecutive instructions.
- Instruction Parallelism (IP): The maximal amount of instructions that a processor can start working on in a single cycle.

The conceptual instructions comprise 4 simple operations that map directly to the pipeline stages. The remainder of this section will refer to the number of stages as s , and assume $s = 4$ as defined by the stages above.

3.1.3.1 Scalar Pipeline

The baseline machine is a straightforward scalar pipeline implementation. The OL, IL and IP equal one cycle; MP equals s . The table below illustrates the operation of this pipeline for a program of 5 instructions:

C^b	IF	DE	EX	WB
1	I ₁	—	—	—
2	I ₂	I ₁	—	—
3	I ₃	I ₂	I ₁	—
4	I ₄	I ₃	I ₂	I ₁
5	I ₅	I ₄	I ₃	I ₂
6	—	I ₅	I ₄	I ₃
7	—	—	I ₅	I ₄
8	—	—	—	I ₅

^b baseline machine cycle

Table 3.2: scalar pipeline principle

The pipeline in this example has a MP of 4. The program saturates the pipeline and obtains maximal efficiency.

Every subsequent cycle, a new instruction enters the pipeline at the IF stage until no more instructions are available. Instructions that are *in-flight*, i. e. in the process of being executed, progress to the respectively following stage in the pipeline. An instruction is fully completed when it leaves the WB stage and therefore the pipeline.

The machine cycle time is in principle determined by the slowest stage in the pipeline: all in-flight instructions must progress to their

respective next stages synchronously. It is therefore desirable to balance the stages in the pipeline such that they require approximately the same amount of time to complete their task.

Based on this limited and theoretical presentation, one might assume that more pipeline stages will always result in higher throughput. This is unfortunately not the case; many limitations apply and deepening a pipeline has diminishing returns. Extensive discussions on this topic are presented in [SL05] and [HP11].

3.1.3.2 Super-pipeline

Super-pipeline designs pursue the pipeline principles with more determination. They divide the pipeline stages into Φ phases which reduces the cycle time by the same factor. They have an **IL** and **IP** of one cycle, and an **OL** of Φ cycles. **MP** equals $\Phi * s$. The table below illustrates the operation of a super-pipeline with $\Phi = 2$ for a program of 5 instructions:

C^b	C^m	IF		DE		EX		WB	
		$\phi 1$	$\phi 2$	$\phi 1$	$\phi 2$	$\phi 1$	$\phi 2$	$\phi 1$	$\phi 2$
1	1	I ₁	—	—	—	—	—	—	—
	2	I ₂	I ₁	—	—	—	—	—	—
2	3	I ₃	I ₂	I ₁	—	—	—	—	—
	4	I ₄	I ₃	I ₂	I ₁	—	—	—	—
3	5	I ₅	I ₄	I ₃	I ₂	I ₁	—	—	—
	6	—	I ₅	I ₄	I ₃	I ₂	I ₁	—	—
4	7	—	—	I ₅	I ₄	I ₃	I ₂	I ₁	—
	8	—	—	—	I ₅	I ₄	I ₃	I ₂	I ₁
5	9	—	—	—	—	I ₅	I ₄	I ₃	I ₂
	10	—	—	—	—	—	I ₅	I ₄	I ₃
6	11	—	—	—	—	—	—	I ₅	I ₄
	12	—	—	—	—	—	—	—	I ₅

^b baseline machine cycle ^m minor machine cycle

Table 3.3: super-pipeline principle ($\Phi = 2$)

The pipeline in this example has a **MP** of $2 * 4 = 8$. The program is unable to extract peak performance as it does not contain a sufficient amount of instructions to saturate the pipeline.

The distinction between phases and stages may seem unnecessary from the viewpoint of instruction throughput: increasing the number of stages would allow an equal decrease in cycle time, and the **IL** remains one cycle in both cases. However, consider two dependent instructions, i.e. one instruction produces a value that is an operand to the other. Evidently, the operand is not produced before the first instruction finishes executing, i.e. leaves the **EX** stage. Therefore, no

two *dependent* instructions may be processed in different phases of the same stage.

At its root, this problem is caused by the increased parallel execution in the machine. The dependencies between instructions do not only determine the order in which they can be executed, but also prohibit their maximal overlap. Problems such as these are referred to as *pipeline hazards*, and their solutions as *pipeline interlocks*. A detailed discussion on pipeline hazards is presented in [SL05]; in the context of this dissertation it suffices to state that pipeline interlocks may pose a limit to software efficiency.

3.1.3.3 Super-scalar Pipeline

Super-scalar pipelines increase *MP* by replicating n scalar pipelines. *IL* and *OL* equal one cycle, and *IP* equals n . *MP* equals $n * s$. The table below illustrates the operation of a super-scalar machine with $n = 3$ for a program of 9 instructions:

C^b	IF	DE	EX	WB
1	I ₁ , I ₂ , I ₃	—	—	—
2	I ₄ , I ₅ , I ₆	I ₁ , I ₂ , I ₃	—	—
3	I ₇ , I ₈ , I ₉	I ₄ , I ₅ , I ₆	I ₁ , I ₂ , I ₃	—
4	—	I ₇ , I ₈ , I ₉	I ₄ , I ₅ , I ₆	I ₁ , I ₂ , I ₃
5	—	—	I ₇ , I ₈ , I ₉	I ₄ , I ₅ , I ₆
6	—	—	—	I ₇ , I ₈ , I ₉

^b baseline machine cycle

Table 3.4: super-scalar pipeline principle ($n = 3$)

The pipeline in this example has a *MP* of $3 * 4 = 12$. Its behaviour is very similar to a super-pipeline and the example in Table 3.3. The pipeline is again not saturated as the *MP* remains higher than the number of instructions.

The super-scalar pipeline is able to process multiple instructions in parallel, i. e. with *full* overlap. This further complicates the problem of dependent instruction execution compared to super-pipelines as an implementation must either ensure that no dependent instructions enter *EX* at the same time, or implement techniques to temporarily halt pipelines that contain instructions waiting for their operands.

OOE The high level of Instruction-Level Parallelism (*ILP*) that may be exploited by super-scalar pipelines gives rise to a complex technique to keep all stages and phases busy: Out-of-Order Execution (*OOE*). Processors that support *OOE* will analyse the data dependencies between multiple instructions of a program stream and schedule them as soon as their operands are available. This technique is intended to execute instructions with no regard for the order in which they were specified.

OOE is an advanced feature and its implementations are highly complex. It is extensively discussed in literature, e. g. [SL05] or [HP11]; in

the scope of this dissertation it suffices to state that OOE cores schedule instructions based on their dependencies rather than on the order in which they appear in a program stream.

3.1.3.4 Super-scalar Super-pipeline

Super-pipelined machines may also duplicate n super-pipelines instead of simple scalar pipelines and combine the features of both designs. The cycle time is reduced, **IL** and **OL** equal one cycle, and **IP** equals n . **MP** equals $\Phi * n * s$. The table below illustrates the operation of a super-scalar super-pipelined machine with $\Phi = n = 2$ for a program of 6 instructions:

C^b	C^m	IF		DE		EX		WB	
		$\phi 1$	$\phi 2$	$\phi 1$	$\phi 2$	$\phi 1$	$\phi 2$	$\phi 1$	$\phi 2$
1	1	I_1, I_2	—	—	—	—	—	—	—
	2	I_3, I_4	I_1, I_2	—	—	—	—	—	—
2	3	I_5, I_6	I_3, I_4	I_1, I_2	—	—	—	—	—
	4	—	I_5, I_6	I_3, I_4	I_1, I_2	—	—	—	—
3	5	—	—	I_5, I_6	I_3, I_4	I_1, I_2	—	—	—
	6	—	—	—	I_5, I_6	I_3, I_4	I_1, I_2	—	—
4	7	—	—	—	—	I_5, I_6	I_3, I_4	I_1, I_2	—
	8	—	—	—	—	—	I_5, I_6	I_3, I_4	I_1, I_2
5	9	—	—	—	—	—	—	I_5, I_6	I_3, I_4
	10	—	—	—	—	—	—	—	I_5, I_6

^b baseline machine cycle ^m minor machine cycle

Table 3.5: super-scalar super-pipeline ($\Phi = n = 2$)

The pipeline in this example has a **MP** of $2 * 2 * 4 = 16$. As could reasonably be expected, its behaviour is very similar to the individual pipeline designs it combines, but requires yet more program instructions to reach peak performance

3.2 MEMORY

This section provides an overview of the main concerns related to memory subsystems of computer hardware. The first section states the requirements for shared memory to allow deterministic computations. The second section gives an overview of many common optimisations intended to reduce the observed memory request latencies.

3.2.1 Coherency

When multiple caches store the same data in a shared memory system, consistency problems may arise. In general, three fundamental

properties must hold for reads and writes to the same memory location.

- *Program order preservation*: A processor P first writes to and subsequently reads from a memory location M. No processor writes to memory location M between the write and read actions of processor P. If the former holds, the result of processor P's read action must yield the value it previously wrote.
- *Coherent view of memory*: A processor P₁ first writes to a memory location M. A processor P₂ subsequently reads from the same memory location M. No processor writes to memory location M between the write and read actions of P₁ and P₂ respectively. If the former holds and both actions are sufficiently separated, the result of processor P₂'s read must yield the value P₁ previously wrote.

A memory consistency model defines when both actions are sufficiently separated. In its most strict form, all changes to memory are always immediately observed by all processors in the system.

- *Sequential Consistency*: Multiple writes change a memory location M. All writes must have a unique order in which they are performed, and all processors must observe the same order of changes to memory location M. In other words, all processors must have the same chronological view of changes to a given memory location.

3.2.2 Optimising Transactions

Optimisations in micro-architectures generally apply to one or more of the four main instruction types [SL05]:

- *computations* perform arithmetic, logical and shift operations on register operands.
- *loads & stores* move data between memory and registers.
- *jumps & branches* control the program flow.
- *others* perform special system functions and operations in the co-processors. In turn, they may be one of the types above from the viewpoint of the co-processor, e.g. computational floating-point instructions are executed by the Floating-Point Unit (FPU).

For raw performance, the most important instruction types are computations and loads & stores. Computations have constant OLs and optimising their execution is — in the context of this dissertation — sufficiently discussed in Section 3.1.

This section focuses on the optimisations that are present in computer memory hierarchies and subsystems to increase the bandwidth (i.e. throughput of information) and decrease the latency. The following sections describe the impact of a single optimisation whilst assuming a relative absence of computational operations, i.e. the assumption holds that performance is strictly bound by memory operations.

Memory Hierarchy

Data caches lower the observed latency for statistically predictable memory requests to layers higher up in the hierarchy. Memory access patterns with high temporal or spatial locality will observe drastically reduced latencies compared to main memory.

Translation Lookaside Buffer

A Translation Lookaside Buffer (TLB) exploits temporal locality in memory access patterns to improve virtual memory address translation speed. Memory requests that hit the TLB will observe a significantly lower latency. Additionally, computer systems often have configurable page sizes, which determines memory page alignment and the maximal number of entries in the buffer.

Memory Prefetching

Data prefetchers further enhance the effectiveness of caches by explicitly observing the memory access patterns exhibited by software. They increase throughput by requesting data ahead of time, before the actual request is made by the program. Upon successful prediction, data will already be cached when it is needed and only the cache latency will be observed. Contemporary prefetching caches are able to detect sequential as well as strided access patterns.

The benefit and efficiency of prefetching data is further enhanced by the low latency of cache memories. Memory requests miss the cache less often and observe significantly lower latencies on average, which in turn increases the throughput of the memory requests in the programs. This is a positive feedback loop and will approach main memory throughput even for programs without temporal locality, but with (sustained) spatial locality.

(No) Write Allocate

Writing to memory must not be done speculatively as this may cause undeterministic behaviour (Section 3.2.1). Additionally, peak memory write bandwidth may be limited by overhead caused by the cache subsystem. WB caches do not typically perform writes directly to main memory, but to cache lines only. This strategy is known as *write allocate* and decreases latency for subsequent read or write operations to the same memory location. If there are no subsequent operations however, memory bandwidth is wasted by fetching the original value. The alternative *no-write allocate* strategy writes values directly to memory and bypasses this overhead. Specialised so-called *streaming* store instructions have no-write allocate semantics and were added to the x86 ISA by Streaming SIMD Extensions (SSE) 2.

SSE 4 also introduces streaming *load* instructions which are likewise optimised for non-temporal data. They store data in specialised memory instead of the main cache to preventing cache trashing. They are currently only supported for Uncacheable Write Combining (USWC) memory however and thus are out of scope for this thesis. The Xeon Phi™ (KNC) does not support streaming load instructions altogether.

Core Duplication

A single thread of execution may not technically be able to saturate the memory subsystem because it is limited by the rate at which a single core may generate requests. Systems with multiple cores may increase load on their memory subsystems by running multiple threads of execution at the same time. This will increase the rate at which requests are received by the memory subsystems and may increase throughput.

Core Registers

A single thread of execution may not need to access the memory system at all. A machine architecture may feature a sufficient amount of registers to store all data that is required. Programs may therefore bypass the memory subsystem completely, in which case no relevant latency nor throughput can be observed.

Core Pipeline Path Forwarding

Data forwarding in the core pipelines causes values to be available to different stages as soon as they are produced. Repeatedly reading or modifying a single memory location may therefore bypass the cache altogether, which will greatly improve observed latencies.

Core Pipeline Depth

As determined by the pipeline design, multiple stages may separate the stage in which memory requests are made from the completion stage. It may therefore take more cycles for an instruction reach the completion stage than it takes for the cache to supply data to a register after a request to the memory subsystem has been made.

When a memory instruction requires the result of a previous instruction to determine the address it must access, it must wait until that instruction is completed and the result is forwarded in the pipeline. A single instruction is therefore unable to observe latencies lower than those inherent in the pipeline design.

Core Pipeline Width

The width of a super-scalar pipeline has an important role in determining the potential exploitation of [ILPs](#) by [OOE](#). Observed memory bandwidth will increase when multiple memory requests present in the instruction stream may be processed in parallel.

Vectorisation

Vector instructions increase performance compared to scalar instructions by operating on multiple scalar values at once. These values must be stored in specialised vector registers which pack multiple scalar values together. Vectorised code is therefore the superior alternative to scalar code for measuring peak cache bandwidth. It is not possible however to use the chasing pointers benchmark for vectorised code as memory addresses are scalar values. The extraction of a single value from packed data requires additional compute cycles and becomes the limiting factor.

Transparent Error Correction

Computer systems may support Error Checking & Correction (ECC) in hardware or software to detect *data errors*, i. e. involuntary changes in stored data. Hardware specifications do not account for overheads that may be caused by correcting errors, but platforms may allow the end user to disable the feature. Notwithstanding the risk of indeterministic behaviour, disabling ECC enables bandwidth benchmarks to be optimised for raw peak bandwidth first. Repeating the tests with ECC re-enabled will then not only provide an estimate for practical peak bandwidth for genuine applications, but also indicate the relative performance cost of the technology.

3.3 MANY INTEGRATED CORE: INTEL[®] XEON PHI[™]

All technical specifications presented in this section are based on public documentation made available by Intel[®]: “Intel[®] Xeon Phi[™] Core Micro-architecture” [Rah13] and “Intel[®] Xeon Phi[™] Co-processor System Software Developers Guide” [Cor14].

The Intel[®] Xeon Phi[™] co-processors code-named KNC are Peripheral Component Interconnect Express (PCIe) extension cards designed to accelerate parallel workloads in a computer system. They integrate up to 61 processor cores and eight memory controllers on a single chip. A high-performance on-die bidirectional interconnect enables communication between all components.

3.3.1 Processor Cores

Each core is in itself a fully functional 64-bit processor including 32-bit support. They have extensive vector processing capabilities as part of the Initial Many Core Instructions (IMCI) extension to its ISA, but do not support OOE, i. e. they are *in-order* cores (cf. OOE, Section 3.1.3.3). A KNC core supports four hardware threads and contains two pipelines: the V-pipe for scalar instructions and the U-pipe for vector instructions¹.

Main memory is governed by eight memory controllers and is designed for high bandwidth rather than low latency. Each core features a L1 data-cache of 32 kB and a L2 data-cache of 512 kB. Additionally, they contain an instruction-cache of 32 kB.

3.3.1.1 Instruction Decoding

The KNC processor core is a barrel processor: it switches every cycle between threads of execution, a technique also known as fine-grained temporal multi-threading. A core supports up to 4 hardware threads. Switching between threads happens in a round-robin fashion. This not only simplifies hardware design, but also ensures that every active hardware thread is assigned an equal amount of processor time.

The instruction decoder is a fully pipelined two-cycle unit, i. e. the decoder has a throughput of 1 Instructions Per Cycle (IPC), but decoding a single instruction requires two cycles. This design artefact corresponds with the PF stage of KNC core instruction pipeline, shown in

¹‘V’ in V-pipe is not for ‘Vector’; it is a legacy name.

Table 3.6. A consequence of dividing this stage in two phases is that no two dependent instructions may be fetched in subsequent cycles.

However, the instructions are not yet decoded and dependency analysis is hence not yet possible. The **KNC** pipeline design solves this conundrum by switching hardware threads every other cycle, as instructions from different threads can not have (direct) data dependencies.

A high-level impact of this design is that at least two hardware threads per core are needed to unlock the co-processor's full potential. When only one thread is assigned to a core, it will switch every cycle to a dummy thread, thereby halving the core's potential. If two threads can not run practically synchronised, e.g. due to different memory access patterns or cache conflicts, additional threads may be beneficial by hiding these accidental latencies.

3.3.1.2 Instruction Scheduling

The instruction scheduler is dual-issue: it can issue instructions to both pipelines in the same cycle. Not all instructions are complementary: pairing rules determine which instructions can be combined. Best-case instruction throughput for a **KNC** core therefore equals 2 **IPC** or 0.5 Cycles Per Instruction (**CPI**).

3.3.1.3 Core Pipeline

Most integer and mask instructions have a one-clock latency. Most vector instructions have a four-clock latency with a one-clock throughput. By supporting four hardware threads, the core facilitates filling the U-pipe with independent vector instructions.

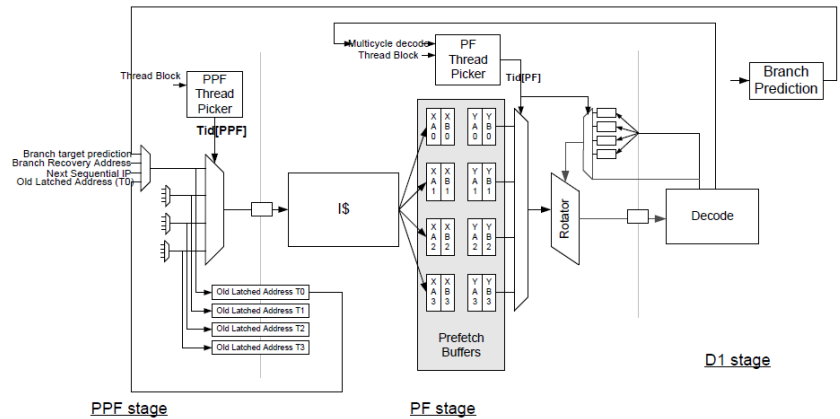


Figure 3.1: **KNC** instruction prefetching and decoding (sourced from [Cor14])
note: *I\$* is a common abbreviation for *instruction cache*

Table 3.6 shows the **KNC** core pipeline, its stages and its phases. In combination with Figure 3.1 it describes how and why the core selects instructions from a different hardware context every cycle and places them in the PF buffer.

As explained in the two previous sections, the core will switch to a different hardware context every cycle in the PPF stage. The PF stage will

stage	phase	operation
PPF	$\phi 1$	thread picker
PF	$\phi 1$	instruction cache lookup
	$\phi 2$	prefetch buffer write
Do	$\phi 1$	thread picker
	$\phi 2$	instruction rotate
	$\phi 3$	decode of of, 62, D6, REX prefixes
D1	$\phi 1$	instruction decode
	$\phi 2$	CROM lookup
	$\phi 3$	Sunit register file read
D2	$\phi 1$	microcode control execution
	$\phi 2$	address generation
	$\phi 3$	data cache lookup
	$\phi 4$	register file read
E	$\phi 1$	integer ALU execution
	$\phi 2$	retire / stall / exception
	$\phi 3$	determination
WB	$\phi 1$	integer register file write
	$\phi 2$	condition code (flag)
	$\phi 3$	evaluation

^a The *Sunit* is an architectural component of the core.

Table 3.6: [KNC](#) Core Pipeline

3.3.2 Memory Organisation

[KNC](#) does not have a L3 data-cache level shared between different execution cores, such as those found in modern Central Processing Units (CPUs). Instead, a global distributed tag directory keeps track of the location of all cached data across the chip. This allows for a given core to fetch data from the L2 data-caches of all other cores, thereby avoiding an access to global memory and the associated performance penalty. In other words, data requests may require access to any and all of the four main memory levels of the [KNC](#) in the following order: L1 data-cache, local L2 data-cache, foreign L2 data-cache, main memory.

Older in-order architectures would traditionally stall the entire core upon any L1 or L2 cache miss. The modern [KNC](#) in-order architecture alleviates this severe performance penalty by only stalling the hardware context when a *load* request misses the cache. The processor will then suspend the hardware context and allow another hardware context to continue. Both the L1 and L2 cache subsystems support up to 38 outstanding read and write requests per core and will only stall an entire core when this limit is reached. Furthermore, the relat-

ively high amount of supported outstanding requests enables liberal memory prefetching.

The following two sections describe both cache levels in further detail, Table 3.7 provides a summary.

Property	L1 Cache	L2 Cache
Coherence	MESI	MESI
Size	32 kB + 32 kB	512 kB
Associativity	8-way	8-way
Line Size	64 B	64 B
Banks	8	2
Access Time	1 cycle	11 cycles
Policy	pseudo LRU	pseudo LRU
Duty Cycle	1 per clock	2 per clock
Ports	Read or Write	Read or Write

Table 3.7: Xeon Phi™ Cache Hierarchy

3.3.2.1 L1 Cache

Each core has a 32 kB L1 data-cache and a 32 kB L1 instruction-cache. They are 8-way associative and cache lines are 64 B wide, which results in 64 associative sets. Banks are 8 B wide, for a total of 2048 banks. The cache replacement algorithm is based on a pseudo-Least Recently Used (LRU) implementation. The load-to-use latency is 1 cycle for integer instructions, vector instructions experience different latencies.

The L1 cache is able to return data out of order — notwithstanding the in-order architecture of the KNC cores — which helps confining the effects of a miss to a single hardware context (also see Section 3.3.1.1).

The L1 cache is subject to the effects of Address Generation Interlock (AGI): address generation is only the second of a total of four phases in the second decode stage, D2. The result of an memory instruction is therefore only available three cycles later. In other words, as a paraphrase of the Xeon Phi™ programming guide: “AGI latency for KNC cores is at least three cycles, i. e. a General-Purpose Register (GPR) value must be produced at least three cycles before it is used as a base or index register in an address computation.” For example, assuming all data is stored in L1 data-cache, the for-loop in listing 3.1 will exhibit three-cycle latencies at best for every read operation.

```
/* assume a[] is stored in L1 cache */
int a[] = {1, 2, 3};
/* loop will observe memory latency as max(AGI, L1) */
for (int i = 0; i < 3; i = a[i]);
```

Listing 3.1: Address Generation Interlock

3.3.2.2 L2 Cache

Each core has 512 kB L2 data-cache. It is 8-way associative and cache lines are 64 B, which results in 1024 sets. As with the L1 cache, the replacement algorithm is based on a pseudo-LRU implementation. The load-to-use latency has a theoretical minimum of 11 cycles. Expected idle load-to-use latency is 80 cycles.

The L2 Cache of the KNC cores is inclusive: all data present in L1 must also be present in L2. Inclusive caches simplify cache coherency implementations (Section 3.3.4) by making it unnecessary to consider L1 caches when searching for the location of cached data.

Streaming hardware prefetchers help reducing perceived memory latencies by selectively loading data into the L2 cache. This technique will increase the cache hit rate for streaming memory access patterns by making data available from cache memory by the time it is needed.

All cores work together in a peculiar way to make a large, shared and global L2 cache. If no cores share any data or code, they will each contribute 512 kB of L2 storage. At the other extreme, when all cores share exactly the same code and data, and run in perfect synchronisation, the same data will be duplicated in every L2 cache. In other words, depending on the workload of the co-processor, applications may perceive an aggregate L2 data cache size of anywhere between 512 kB and 30.5 MB (assuming the maximum of 61 cores).

3.3.2.3 Main Memory

The eight memory controllers of the KNC are spaced evenly across the ring interconnect. A single controller supports two memory 32-bit channels which support a transfer speed of 5.5 GT/s. The aggregate peak bandwidth for the complete KNC processor is therefore 352 GB/s.

3.3.3 Page Tables

Applications for modern computer Operating Systems (OSs) run in a virtual memory address space. This technique allows programs to run oblivious to any other software on the system as all computer memory appears to be available for its own purposes. The OS transparently maps the virtual addresses to physical addresses. Performing this mapping for every single memory access would be a very expensive operation in terms of performance.

In order to reduce the memory overhead in address translation bookkeeping, the OS divides the full address space into smaller unitary regions, i.e. memory pages. This technique avoids the need to store a mapping for every single address by associating complete pages with applications. Multiple page sizes are typically supported; larger page sizes allow for larger memory regions to be mapped, but result in more significant fragmentation of the physical address space.

The Memory Management Unit (MMU) of modern processors also helps in reducing the lookup cost by providing a *page table* and a *TLB*. The table may store the virtual-to-physical memory mappings that are currently in use by the OS. The TLB stores the address mappings of the most recently accessed pages and significantly accelerates the translation process. These optimisations in hardware are typically very efficient, such that the cost of address translation is no longer observable.

However, on a TLB miss, i.e. when the TLB does not contain the requested mapping, a lookup has to be performed in the page table, which incurs a performance penalty.

The KNC supports page sizes of 4 kB and 2 MB. It features a hierarchical page table of four levels, i.e. a four-level page table walk is performed upon a TLB miss. Table 3.8 further details the paging structures of the KNC co-processor.

TLB	Page Size	Entries	Associativity	Maps
L1 data	4 kB	64	4-way	256 kB
	2 MB	8	4-way	16 MB
L1 instruction	4 kB	32	4-way	128 kB
L2 (data)	4 kB, 2 MB	64	4-way	128 MB

Table 3.8: KNC Page Tables

3.3.4 Coherency Protocols

The KNC is a multi-processor system with private caches for each processor and must therefore implement a solution to provide memory coherency semantics (Section 3.2.1). Both cache levels of the KNC cores use the standard MESI protocol [PP84] (overview in Table 3.9) to ensure data consistency across all cores.

The *owner* state found in the MOESI protocol allows a single cache to store the ‘master’ copy of a cache line. The owner of the cache block may then modify the data without restrictions. When another core requests the data, it may be forwarded directly by its owner, instead of writing back the changes to main memory.

The lack of an owner state in the KNC coherency protocol incurs potential performance limitations. The next section discusses Tag Directories (TDs) which help solve this problem.

3.3.4.1 Tag Directories

KNC uses a TD which is responsible for the global coherency between the different L2 caches. It implements the GOLS₃ protocol (detailed in Table 3.10), which complements MESI and counteracts the potential performance degradation due to the lack of an owner state.

The TD is able to identify the location of cached data across all L2 caches on the chip. In the event of a (local) L2 cache miss, the core will use the TD infrastructure to request data. The TD is then responsible to request the data from another core’s L2 cache or from main memory when this also fails. The TD itself is physically distributed across the ring interconnect to amortise lookup costs.

State	Definition
M	Modified. Cache line is updated relative to main memory. Only one core may have a given cache line in M-state at a time.
E	Exclusive. Cache line is consistent with memory. Only one core may have a given line in E-state at a time.
S	Shared. Cache line is shared and consistent with other cores, but may not be consistent with main memory. Multiple cores may have a given line in S-state at the same time.
I	Invalid. Cache line is not present in this core's L2 or L1 cache.

Table 3.9: MESI States

State	Definition
GOLS	Globally Owned, Locally Shared. Cache line is present in one or more cores, but is not consistent with main memory.
GS	Globally Shared. Cache line is present in one or more cores and consistent with memory.
GE/GM	Globally Exclusive/Modified. Cache line is owned by one and only one core and may or may not be consistent with memory. The TD does not know whether the core has actually modified the line.
GI	Globally Invalid. Cache line is not present in any core.

Table 3.10: GOLS₃ States

This chapter describes the pipeline model: a performance model to assess the running time and efficiency of genuine applications on fine-grained hardware. It revolves around the concepts of Thread-Level Parallelism (TLP) and instruction latency hiding. The model strives to keep a good balance between the simplicity of the ubiquitous O-Notation [Knu97] and the precision provided by cycle-accurate simulations.

The content in this chapter is chiefly based on a internal and early draft on the pipeline model. The model itself is still work in progress and subject to change; this dissertation presents a minor reinterpretation of the model, as it was originally developed for Graphics Processing Unit (GPU) architectures. Central Processing Unit (CPU) architectures like the Many Integrated Core (MIC) use techniques different from those used in GPU architectures to exploit fine-grained parallelisms.

Section 4.1 discusses the general philosophy of the pipeline model and the trade-offs it chooses to make. The following section, 4.2, presents an overview of the model and the concepts it comprises. Section 4.3 continues with a number of examples that apply the pipeline model. Section 4.5 presents a discussion on the mapping of the model to CPU and MIC architectures. The most apparent difference between CPUs and GPUs is the scale on which they implement Fine-Grained Parallelism (FGP) and exploit Instruction-Level Parallelism (ILP), which is reflected in their programming models. The chapter concludes with a short presentation on related work in Section 4.5.

4.1 THE NEED FOR A FINE-GRAINED PERFORMANCE MODEL

The O-Notation is an asymptotic representation of a function, expressing the relation between the input size and the running time of an algorithm. It is the quintessential performance model and a great tool to compare algorithms as mathematical entities, because it is not concerned with effective execution times.

Cycle-accurate simulations define an opposite end in the performance model spectrum; they calculate the exact running time of a program, but are specific to a single hardware design. They require full knowledge about the execution environment and the program being executed.

The pipeline model relies on empirical information gathered by means of micro-benchmarks to model the execution environment, and on analysis about the operational composition of a software application. It combines this information to provide an estimate on the efficiency of the latter when it is executed on the former. The model is able to provide information with different levels of detail, which supports an incremental approach to software efficiency optimisation. The following subsection motivates the use for micro-benchmarks compared to its alternatives.

Using parameters from both software and hardware, the pipeline model estimates time needed for a processor to execute a program.

4.1.1 *Online vs. Offline Efficiency Analysis*

For Fine-Grained Parallelism workloads, processor hardware is not able to provide end-users with a sufficiently detailed trace of execution, i.e. complete information on how a program is effectively and precisely executed. The most precise information available is provided by the aptly named hardware performance counters, which record the prevalence of certain hardware events, such as sustained cache misses, executed floating-point instructions, etc.

The amount of observable hardware events is most often too large to implement separate performance counters for all different event types. In practice, a counter must be first configured to observe a desired hardware event. Software profiling applications may automate the measurement of a large set of hardware events by changing the hardware performance counter configuration between multiple runs of an application. However, these counters are local to certain hardware subsystems, but global from the viewpoint of an application. I.e. different threads may increase the count of hardware events of interest. Manual code instrumentation is required — but not always possible — when events must only be recorded for a certain portion of the program.

Hardware performance counters remain a great tool for software efficiency analysis, but are highly impractical to use within a limited scope. They are not able to indicate which code is exactly responsible for which event, except in highly controlled environments. They are unable to give insight as to which code segments cause what kind of performance penalty, and the degree

A performance model is therefore a better means to estimate performance, efficiency and overhead for a specific combination of hardware and application. It enables a more detailed understanding of the performance of fine-grained parallel architectures with less effort.

4.2 OUTLINE OF THE PIPELINE MODEL

The pipeline model is based on abstract representations of both the application software and the processing hardware in order to maintain generality. The abstract representations may be individually interchanged to predict performance for different software and hardware combinations.

The following sections present an elaboration on the three major components of the pipeline model:

- A computational and a memory pipeline model the hardware.
- Hardware performance is characterised by effective instruction latencies and effective instruction throughput.
- Software is characterised by its instruction composition and the contexts in which they are executed.

4.2.1 *The Pipelines of the Pipeline Model*

In its current state, the pipeline model uses two conceptual pipelines that simulate the execution of a program. The computational pipeline processes *program instructions*; the memory pipeline processes *memory*

transactions. Memory transactions are generated by memory instructions. Figure 4.1 illustrates the global operation of the pipeline model and the interaction of its conceptual pipelines.

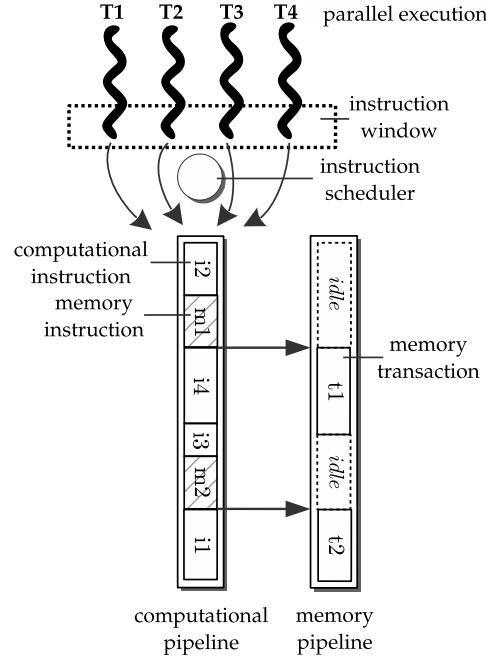


Figure 4.1: the conceptual pipelines of the pipeline model

4.2.2 Hardware Performance

To model optimal hardware performance, the pipeline model requires estimations of the general performance metrics defined in Section 2.1. It defines *issue* and *completion* latencies which are closely related to respectively *throughput* and *latency* as defined in the previous chapter. Additionally, it requires a quantification of static constraints on concurrent execution, e.g. number of hardware threads per processor core. Latency hiding and instruction overlapping is modelled according to the principles of fine-grained TLP which closely resembles how GPUs operate.

The three hardware parameters are formally defined as follows:

Definition 6 (λ — issue latency)

λ , the issue latency, describes the rate at which an instruction scheduler can issue independent instructions to the core pipelines. It is the minimal ratio of unit time to completed instructions.

Definition 7 (Λ — completion latency)

Λ , the completion latency, describes the chronological distance between issuing an instruction and the availability of its result to subsequent instructions. It is the minimal ratio of unit time to completed interdependent instructions.

Definition 8 (concurrent execution constraints)

Concurrent execution constraints limit the potential concurrency of a processor. They are static limitations imposed by how an application is mapped to the hardware.

An informative observation is that these parameters are similar to those defined by Jouppi [JW89] (also see Section 3.1.3):

- λ effectively combines the Issue Latency (**IL**) and Instruction Parallelism (**IP**) parameters for instructions; **IP** is a static property of the machine, while **IL** may vary according to run-time restrictions. Simply put, $\lambda = \frac{\text{IL}}{\text{IP}}$.
- Λ is equivalent to the Operation Latency (**OL**) of an instruction, but may vary depending on the *context* it is executed in, as explained in the following subsection.
- The concurrent execution constraints correspond with the Machine Parallelism (**MP**) parameter, but allows for static software-imposed restrictions that may limit the **MP**. E. g. the total number of registers required by a **GPU** work unit may cause one or more instruction pipelines to be unavailable.

4.2.3 Software Efficiency

The instruction interdependencies of a software application determine the concurrency potential of an application. Interdependent instructions have to be serialised, i. e. can not be executed concurrently, as the result of one expression is required as an operand for the other. Other instructions may be executed concurrently without affecting program semantics. An instruction dependency graph is a convenient representation of all instruction interdependencies in a program. It is a Directed Acyclic Graph (**DAG**) and can therefore be topologically sorted. The path between the root and its most distant leaf is the *critical path* and describes the ultimate limitation in concurrency.

The contexts allow for different latencies when software inefficiencies expose hardware bottlenecks, such as, uncoalesced memory access, partition camping, etc.

Definition 9 (instruction dependency graph)

*An instruction dependency graph is a Directed Acyclic Graph (**DAG**) that expresses the data dependency relationships between the instructions of a program.*

Definition 10 (instruction context)

A context models the variability of the completion latency of an instruction. It parameterises an instructions' completion latency.

4.3 PIPELINE SIMULATIONS

The operation of the computational pipeline depends on the parameters of the model and is able to identify different causes for limited performance or efficiency. This section aims to provide the reader with an intuitive understanding of the computational pipeline operation by means of three examples.

Maximal Performance

Figure 4.2 shows the execution of a program with 5 threads that comprise a stream of interdependent instructions each. All instructions have an equal λ , and $\Lambda = 4 * \lambda$. The available TLP, defined as c , also equals 4.

The number of threads is greater than both the Λ of any instruction and c . As instructions from different threads can not be data-dependent, the machine is able to start work on a new instruction as soon it is able to, i. e. as dictated by the λ of the instructions.

The instruction throughput is optimal, and the total execution time T can be trivially calculated by $T = n * \lambda + (\Lambda - \lambda)$, where n is the total number of instructions executed. The second term represents what is called the *drain* time of the pipeline.

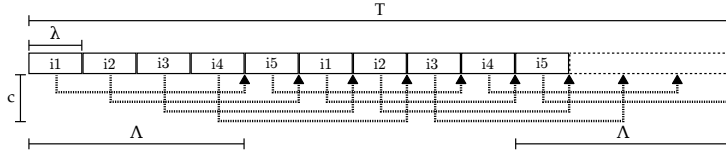


Figure 4.2: computational pipeline: maximal performance

Limited Latency Hiding

Figure 4.3 shows the execution of the program defined in the previous example, but restricts its execution to only 3 threads. This consequently reduces c to 3, as the program does not present sufficient TLP to fully exploit the hardware capabilities.

Note that these changes do not *directly* cause a loss in efficiency. The cycles that are *lost* are a consequence of insufficient *latency hiding*, i. e. $\lambda * c < \Lambda$. The visual scheme clearly shows that an additional λ is added for every three instructions. This is a recurring pattern caused by insufficient latency hiding and occurs every time a dependent instruction must be scheduled, i. e. $\lfloor \frac{n-1}{c} \rfloor$ times. The incurred penalty for every time this pattern occurs is simply $\Lambda - c * \lambda$ ¹.

The total execution time can thus be calculated as follows: $T = n * \lambda + (\Lambda - \lambda) + (\Lambda - c * \lambda) * \lfloor \frac{n-1}{c} \rfloor$, i. e. *peak instruction throughput* plus *drain time* plus *lost cycles*. Solving this equation results in $T = 6\lambda + 3\lambda + 1\lambda * \lfloor \frac{5}{3} \rfloor = (6 + 3 + 1)\lambda = 10\lambda$.

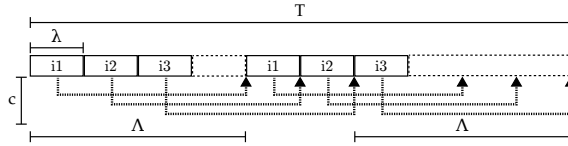


Figure 4.3: performance limited by Instruction-Level Parallelism

The software efficiency of this program can be improved by adding another thread: the additional instructions will take the place of the lost cycles, i. e. the machine is capable of executing *more work* in the *same time*.

¹Note that this penalty is equal or less than zero for optimal programs, i. e. this formula is only defined on \mathbb{N}

Overcommitting

Figure 4.4 illustrates again the execution of the program in the first example, but on a machine where c is limited to 3 by *concurrent execution constraints*. This results in the same execution pattern as in the previous example and its execution time may be calculated by the same equation: $T = 10\lambda + 3\lambda + 1\lambda * \lfloor \frac{9}{3} \rfloor = (10 + 3 + 3)\lambda = 16\lambda$.

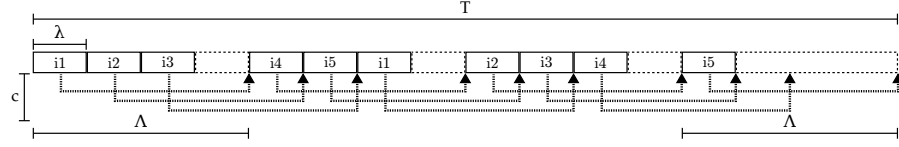


Figure 4.4: performance limited by concurrent execution constraints

However similar the execution may be, its *cause*, and therefore its *solution*, is different. If the same workload would be distributed on only three threads, the execution time would remain unchanged. However, when redistributing the same workload across *more threads* lifts or reduces the concurrent execution constraints, c will be restored to its optimal value and no more latencies will be lost.

It should be noted that such changes often result in a trade-off that may introduce additional computations in the program as a whole. The actual effectiveness and efficiency of such changes also depends on the program itself.

4.4 THE PIPELINE MODEL FOR CPU AND MIC ARCHITECTURES

The scheduling logic of the current iteration of the pipeline model only considers **TLP** to provide **ILPs** and latency hiding. This is highly similar to the effective instruction scheduling mechanisms employed by **GPU** architectures, but drastically different from **CPU** architectures. The most important differences between instruction scheduling in **CPUs** and **GPUs** are condensed in the following subsections. This section concludes with a proposal on how to slightly reinterpret the pipeline model to support different architectures.

4.4.1 CPU Dissimilarities

Most contemporary and high-performance designs consider only one or two threads *at the same time* when selecting instructions to execute. Additionally, they support far fewer threads in hardware compared to **GPUs**, and switching between different software threads requires a so-called *context switch* which is a relatively costly operation.

Instead, these architectures take many instructions from a single thread in consideration, analyse their data dependencies, and start executing them when their operands are available, i.e. they exploit the **ILP** that is present *within* a single thread of execution instead of the **ILP** that directly follows from **TLP**.

4.4.2 MIC Dissimilarities

The Xeon Phi™ is a processor that takes a middle ground: it is able to switch between four threads of execution on a single core with

no extra cost. However, its in-order design implies that it does not attempt to execute instructions as soon as their operands are available. It further complicates its instruction scheduling mechanism by using an intermediate buffer in which instructions from different threads are stored.

Chapter 3 includes the full specifications of MIC architecture; in the context of this section it is sufficient to state that the latter technique allows the core to execute different threads *asynchronously*, i.e. the core efficiency is maintained even when it is executing a ‘slow’ and a ‘fast’ thread.

4.4.3 A Simple Solution

The previous subsection introduced details (Chapter 3 provides many more) in a small space, not to confuse the reader¹, but to illustrate the intricate nature of effective instruction scheduling mechanisms.

All instruction scheduling mechanisms have ultimately the same goal: provide the execution core and its pipelines with as much instructions as possible. This high-level process can be conceptualised as an instruction window that contains the instructions that may be scheduled at a single point in time. Figure 4.1 already includes this slight modification to the pipeline model; the difference with the original image is but the addition of the instruction window.

For machines geared towards the exploitation of TLPs, this window will contain a very small amount of instructions of many different threads. For machines geared towards the exploitation of intra-thread ILPs, this instruction window will contain a large amount of instructions of a single thread.

The data-dependency graph of the instructions that are present in the instruction window will then ultimately determine which instructions may be effectively selected for execution. Out-of-Order Execution (OOE) machines may do so without further restrictions. In-order machines may be modelled by adding the order in which instructions appear in a single thread as a dependency in the dependency graph.

4.5 RELATED WORK

The pipeline model is partly inspired by earlier work on General-Purpose computing on GPU (GPGPU) modelling by Hyesoon et al. in [HK09] and [Sim+12]. This dissertation presents an initial attempt at applying the model to the Xeon Phi™ core architecture. Related research on modelling different aspects of the Xeon Phi™ is available and combining these different ideas may culminate in a pipeline model that is also representative of the Xeon Phi™ as a whole.

SCAT Model

Fang et al. present the Symmetric Cores and Asymmetric Threads (SCAT) model in [Fan+14]. The model is specific to the Xeon Phi™ and provides a software developer with a highly simplified view of the co-processor.

¹The author does not confirm nor deny the accuracy of this statement.

Inter-Core Communication

Ramos and Hoefler [RH13] presented a model for communication in Symmetric Multi-Processing (SMP) systems in combination with a case study on the Intel® Xeon Phi™. They model the interaction between the different cores and the impact of the cache coherency system.

Energy Efficiency

The pipeline model only focuses on software efficiency compared to hardware peak performance. As discussed in Chapter 2, energy efficiency is becoming increasingly efficient. Shao and Brooks have presented an energy model [SB13] of the Xeon Phi™.

SUMMARY

This chapter discussed the pipeline model and proposed an interpretation for CPU and MIC architectures. The model requires several parameters to operate, which can be obtained by micro-benchmarking the hardware. It also requires the empirical exploration of the hardware to identify different kinds of inefficiencies that incur a significant performance penalty. These inefficiencies introduce variable latencies which can be accounted for by the pipeline model's concept of contexts.

The following chapter will present a benchmark framework that facilitates the implementation of (micro-) benchmarks and as well as their use for empirical hardware exploration. Chapter 6 discusses the design of efficient and relevant micro-benchmarks that have the dual purpose of extracting the pipeline model parameters from the hardware, and characterising its behaviour. Chapter 7 continues to discuss the results of running the benchmarks of the previous chapter on both a Xeon® CPU and the Xeon Phi™ co-processor.

This chapter introduces the Analyze Diverse Hardware Demeanors (ADHD) benchmark framework¹. The main design goal of the framework is to provide a future-proof and portable interface for the development of relevant (micro-) benchmarks. It is specifically developed for this thesis and focuses on different key areas than benchmark suites available today, such as BenchIT [Juc+03] or lmbench [M+96].

It considers benchmarks in their most generic form: measuring & reporting the execution time of some payload code, whose behaviour is controlled by a set of variables, i.e. the benchmark configuration. ADHD provides individual abstractions for the concepts of measuring & reporting, payload code execution and automated execution of different configurations. The result is an extremely simple interface to effectively run a benchmark for arbitrary amounts of configurations.

Section 5.1 presents the general ADHD philosophy and its short and long term design goals. The following section, 5.2, introduces ranges as an abstraction for all possible configurations of a benchmark. Section 5.3 presents benchmarks as a specialisation of ranges. Then follows a discussion on the implementation of the multi-threaded benchmark class in Section 5.4. Section 5.5 concludes with a discussion on the importance of the *accurate* collection of timing information.

A generic, reusable and future proof set of abstractions for highly parameterised benchmarks with automated support for parameter sweeping.

5.1 PHILOSOPHY OF THE FRAMEWORK

One of the major motivations for portability lies with the radical differences the Xeon Phi™'s successor will bring in 2015 (see Section 2.6). ADHD is written in portable, strictly standard C++11 and leverages many language features to provide the end-user with a very straightforward interface to implement. The high-level language implementation respects both components of the Dynamic-Static Interface (Section 3.1.2). This enhances the relevance of the code and results to genuine applications as the effects of the complete software stack on performance is taken into account.

A long term goal is to gradually broaden the scope of the framework and allow the implementation of more generic software. The extensive hardware characterisation abilities of the framework could then assist the developer to implement highly efficient software.

The current implementation of the framework limits its scope to what is directly relevant to this dissertation. It supports straightforward and low-effort means to implement highly parameterised, efficient micro-benchmarks and the means to reuse as much code as possible. It includes high-precision measurements, is fully standard compliant and has been tested on many different platforms already.

¹The ADHD source code is available on the following URL: <https://github.com/rdewaele/adhd/>. The discussion presented in this chapter mainly concerns the code in the *benchmarks* folder.

5.2 RANGE ABSTRACTION

Highly configurable benchmarks have many parameters by definition. Characterising hardware using such benchmarks corresponds with running the same benchmark many times while slightly varying its parameters, e.g. a memory latency test for an increasing range of data. [ADHD](#) introduces *ranges* to automate this process and to combine multiple parameters into a single iterable entity.

The essential operations that a range must support are defined by the *RangeInterface* class. It is principally designed to automate support the *range-based* for-loops introduced by C++11 to provide an abstraction to how ranges progress.

Implementations of the interface must therefore specify their lower and upper bounds, forward iteration semantics, i.e. how to progress from the minimal to the maximal value, and how to reset the range to its initial state. The following subsection discusses how different ranges can be composed. The subsequent section describes two generic implementations of the *RangeInterface* already provided by [ADHD](#).

5.2.1 Composition

It is desirable for benchmarks to support multiple ranges in their parameters, e.g. a memory latency test for an increasing amount of data using a variety of access patterns. This adds an extra dimension to the configurability of a benchmark, as the number of possible configurations equals the product of the respective sizes of all ranges.

The *RangeSet* class accepts implementations of the *RangeInterface* in its template arguments and represents their composition. I.e. it is a heterogeneous collection of ranges. Incrementing the set will progressively increment the ranges it comprises in the same order as they were supplied, but a subsequent range will only be incremented when the current one is depleted. This procedure will eventually generate all possible combinations, but requires overflow semantics from the ranges it composes.

Overflow semantics can not be implemented directly by the *RangeSet* class, as a range value should be reset when an increment *would* cause the current value to become greater than the maximum. However, the *RangeInterface* defines predicates that indicate whether a range is at its minimal or maximal value, which is employed by the *RangeSet* to detect whether it should increment the next range it comprises.

5.2.2 Steppers

In order to cover most use cases for simple ranges, [ADHD](#) provides two generic implementations of the *RangeInterface*:

- The *AffineStepper* class provides a common and generic semantic for forward iteration, the affine function: $f(x) = a * x + b$. The affine stepper may therefore wrap any types for which multiplication and addition are defined.
- The *ExplicitStepper* is essentially an ordered set of values with range semantics, i.e. it behaves as a circular list. The *ExplicitStepper* avoids the need for mapping non-arithmetic types to

arithmetic types and back and the associated overheads and accidental complexities, such as printing the range.

Iterating through different configurations may require a mixture of affine, explicit or even custom steppers. [ADHD](#) provides the *RangeSet* template class to compose different steppers.

5.3 BENCHMARKS ARE RANGES

A straightforward benchmark implementation could consist of a set of nested for-loops that iterate over all possible configuration ranges and that measure the execution time of the payload code in the innermost loop. This approach has two major downsides: it involves manual duplication of code and benchmarks can not be easily composed. The former might be tolerated to a certain extent¹, but the latter poses a strict limitation on code reuse.

For example, a multi-threaded benchmark may parameterise the payload code it should run, while increasing the amount of threads between executions. Client code does not have control over when this progression happens, nor access to the iteration variables defined by the benchmark, i. e. the current number of threads.

To solve this problem, [ADHD](#) opts to provide range semantics for benchmarks as well. Benchmarks may implement additional behaviour when progressing to a next value in one of their associated ranges. This allows implementations to decide when to progress to the next stage of the benchmark they extend. This feature also enables loop inversion, i. e. a benchmark implementation may decide to progress through all configurations of its parent first, before it itself progresses to its next configuration, before repeating the process.

There is an important difference between benchmarks and range steppers however, and that is in how they compose. Steppers do not associate behaviour when iterate to their next value, unlike benchmarks.

The similarity between benchmarks and ranges is exploited by the *BenchmarkInterface* class, which inherits the range interface and adds a *run* method. A simple one-shot benchmark interface can be trivially implemented by inheriting from a range implementation with a constant value, in addition to the *BenchmarkInterface* itself. A more interesting implementation is that of the abstract threaded benchmark class, which provides many means to facilitate the implementation of multi-threaded benchmarks.

5.4 MULTI-THREADED BENCHMARKS

Multi-threaded micro-benchmarks are straightforward to implement and extend. The current implementation is based on pthreads [[BF96](#)] for maximal portability and control of behaviour. It supports multiple thread affinity assignment schemes for increased control of how the benchmark load should be distributed across a system. The burden of thread synchronisation is lifted from the end-user by the definition of a multi-phase protocol which covers a vast variety of possible applications.

¹The author has a strong aversion of such code.

5.4.1 Execution in Phases

The *ThreadedBenchmark* class replaces the single run method of the basic interface with five methods that implement an execution protocol of five phases: *init*, *ready*, *set*, *go* and *finish*. The only phase that must be implemented is the *go* phase, which corresponds with the regular run method of a simple benchmark. The other phases serve different purposes; an overview:

1. *init*: The first phase will be executed by only one thread of all threads associated with the benchmark instance. Initialisation steps that must be executed before all threads become active should be executed in this phase, e.g. shared memory allocation.
2. *ready*: The second phase will be executed by all threads associated with the benchmark instance. It caters for per-thread or thread-local initialisation.
3. *set*: The third phase is the warm-up phase. Some benchmarks may want to execute their payload code before measuring its execution time in order to eliminate variance in the timing results.
4. *go*: The fourth phase executes the effective benchmark code.
5. *finish*: The fifth phase mirrors the first. It is executed by only one thread and may collect and print timing information, free resources, etc.

All phases run synchronised by means of pthread barriers. While adequate for most intents, barriers have a limited resolution by which they synchronise threads, i.e. the spread in time of all threads passing a barrier is relatively high. The *go* phase employs a spin-barrier in order to improve the accuracy of the timing information it may provide.

5.4.1.1 Spin-Barrier

The spin-barrier technique involves a shared integer that is initialised to the total number of threads, and single ‘master’ thread that will give the final *go* signal. When the shared integer is initialised, all non-master threads decrease it by one, and start repeatedly reading the value until it drops to zero. The master thread first waits for the shared value to become one, before decreasing it and starting execution of its payload code. As the master thread sets the shared value to zero, all other threads will also start running their respective benchmark code as the change is propagated throughout the processor.

The spin-barrier technique requires atomic operations, i.e. operations that are guaranteed at the hardware level to preserve memory consistency (see Section 3.2.1). This employs the underlying hardware’s cache coherence protocols to communicate a change throughout the whole processor and is as such a much more accurate method to synchronise different threads. The downside to spin-barriers is that they are blocking operations and will waste many compute cycles while reading the shared value. This prevents them not to scale beyond the thread-level (also see Figure 5.1) Machine Parallelism (MP) offered by a processor.

5.4.2 Synchronised Thread Start

As discussed in the previous section, ADHD provides two means to synchronise threads. For the benchmark payload code it is important that different threads start execution within the smallest possible time frame for two reasons. The first is that the overhead in thread communication is included in the global timings, i. e. the time from when the first thread started to when the last thread finished. The second reason is that the different payloads will have less overlap in their execution, which may result in drastically different thread and/or instruction scheduling. The latter does not only make the results more imprecise, it also increases the variation of the results in different trials.

The Xeon Phi™ allows for slightly more precise synchronisation using the spin-barrier method, compared to the barrier method. Figure 5.1 shows the amount of cycles that have passed between starting the first thread and the last thread as the number of total threads increases. (Please note the logarithmic scales of the Y-axis.) The results were obtained on a 61-core Xeon Phi™ and the threads were pinned in a round-robin manner across different cores. Figure 5.1a and 5.1b show the timing information and its (absolute) standard variation obtained using the spin-barrier method and the barrier method respectively. Figure 5.1c combines both graphs into one, and removes the standard deviation plots, to better illustrate the difference between the two methods.

Two interesting properties emerge from the plots in Figure 5.1:

- The spin-barrier method results in a significantly lower variation of the measurements and a tighter spread. It is a strictly superior synchronisation method.
- The vastly increased precision of the spin-barrier method shows that a severe penalty is introduced when the number of threads exceeds the thread-level machine parallelism. The difference in timing is almost three orders of magnitude. This simple benchmark can therefore not only measure which synchronisation methods are best used on different platforms, but also effectively show the available amount of thread-level machine parallelism.

5.5 TIMING INFORMATION

Precise timing information is very important and held in high regard by the framework. It supports measuring cycle counts using the RDTSC instruction and provides a simple interface to hardware performance counters. Hardware performance counters are not standardised and their usage differs on many processor technologies. The Performance Application Programming Interface (PAPI) [CM13] is a software package that presents a unified interface for performance counters. ADHD provides a simple and straightforward C++ wrapper for PAPI. (Also see Sections 4.1 and 6.1.2 for more information about performance counters).

In essence, the most important factor for obtaining correct timing information is how it is implemented in the benchmark payload code. For single-threaded benchmarks, this is the only factor. For multi-threaded benchmarks, it is important that all benchmark threads start

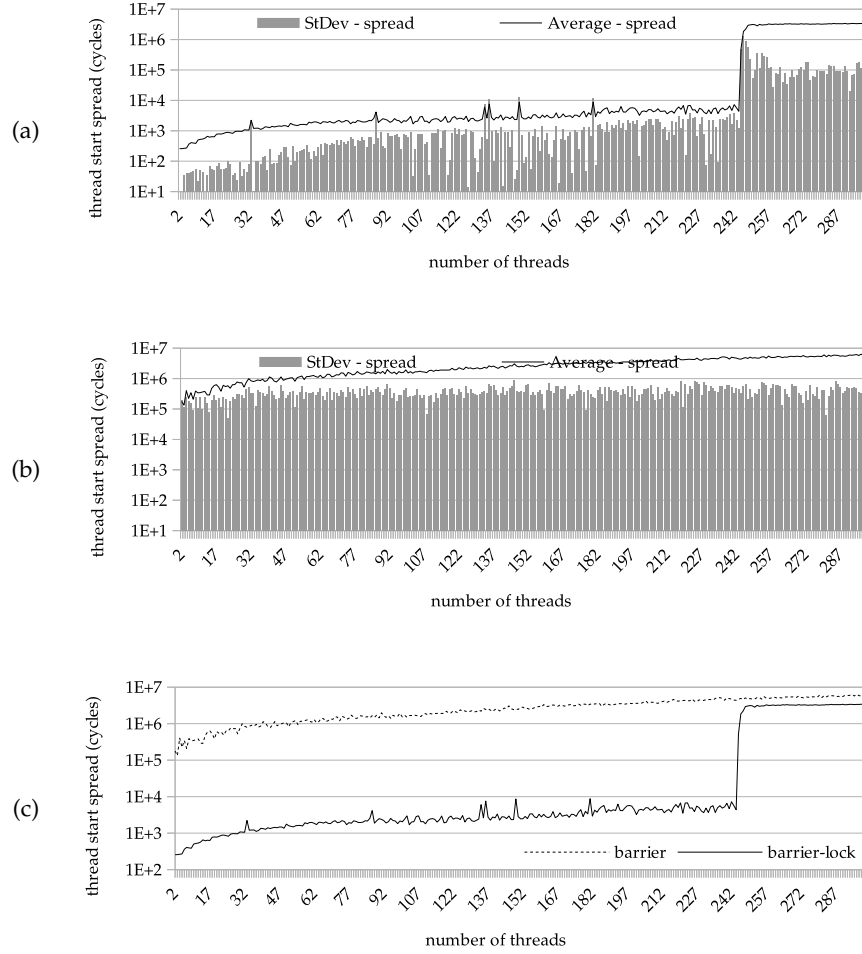


Figure 5.1: Xeon Phi™ thread synchronisation

at the same time to ensure that all workloads interfere as much as possible and that the accidental variance in timing is thereby reduced. The previous section discussed how the ADHD multi-threaded benchmark solves these problems.

5.5.1 Reporting

The principal ‘run’ method of any benchmark accepts a lambda function as an argument. This function serves as the timing callback and will be called every time a single benchmark configuration is executed. The argument of the lambda function itself must implement the *Timings* interface, which defines three simple operations that may print results, i.e. any class that implements this interface declares itself to be printable as comma separated values and as human readable values. The former will typically be used for post-processing while the latter may be used to provide immediate feedback to the benchmark operator.

SUMMARY

This chapter presented the ADHD benchmark framework and the abstractions it provides. It does not only simplify the implementation of benchmarks, but also allows them to be used directly for hardware characterisation by sweeping their parameters.

The following chapter will discuss the design of multiple micro-benchmarks, all of which benefit from the abstractions provided by ADHD.

Designing micro-benchmarks in a high-level language involves more components than manually writing machine code and introduces additional complications to the design process. For example, optimising compilers perform many transformations on their input and may remove redundant operations. Micro-benchmarks may however rely on redundancy in their instructions to assess hardware performance characteristics. The philosophy of creating benchmarks that are representative for genuine applications requires these optimisations to be performed, so great care must be applied to ensure correct semantics for the micro-benchmarks.

The micro-benchmarks will isolate different hardware components on the micro-architecture level and estimate their impact on *throughput*, *latency* and *bandwidth* properties of performance. As these measurements will yield values in the range of just a few cycles, high-precision timing information is required. The following section discusses two means to obtain timing information of extreme precision. All benchmarks are implemented in the Analyze Diverse Hardware Demeanors (ADHD) framework (Chapter 5) and this section will only focus on the design of the benchmark payload code.

The qualitative understanding of micro-architectures is applied to design benchmarks that analyse the impact of individual micro-architectural features. The benchmarks are developed in a high-level languages, which incurs additional complexities introduced by the presence of a compiler.

6.1 OBSERVING ELAPSED CYCLES

Observing wall clock time is a straightforward way to estimate performance, e.g. a program may run for one minute to process 1 GB of data. Wall clock time is however very coarse and therefore inadequate to obtain the cycle-accurate measurements needed for micro-benchmarks.

Processor designs typically include a number of implementation-specific means to observe the behaviour of themselves and the software they are executing. This section discusses two main alternatives for high-precision observation of the low-level behaviour of contemporary high-performance processor architectures.

6.1.1 Time Stamp Counter

The Time Stamp Counter (TSC) is an x86 implementation feature introduced by the first Intel® Pentium® processor in 1993. The TSC tracks the number of elapsed processor cycles since the last processor reset. Most if not all x86 implementations since include a TSC, making it a very convenient source of exact timing information. The Read Time Stamp Counter (RDTSC) instruction returns the current value of the TSC in the EDX:EAX registers. Both registers are 32 bit wide; EDX:EAX is their concatenation, with the most significant bits in EDX. Instrumenting code with calls RDTSC is a precise method for measuring the number of cycles that have elapsed between two program code locations.

```
uint32_t low, high;
asm volatile ("rdtsc":"=a"(low), "=d"(high));
```

```
uint64_t cycles = low | ((uint64_t) high << 32);
```

Listing 6.1: Reading the Time Stamp Counter

Listing 6.1 illustrates how the counter can be accessed from C or C++ using fixed-width integer types and in-line assembly code. The fixed-width integers correspond to the EAX and EDX registers. The *asm* keyword introduces the in-line assembly code and the *volatile* keyword informs the compiler that the instructions must not be moved from where they appear in the code. The *rdtsc* instruction is then followed by two output operands: *=a* and *=d* store the values contained by respectively RAX and RDX in their argument.

6.1.2 Hardware Performance Counters

Section 4.1.1, introduced hardware performance counters. They are a set of special-purpose registers to count hardware-related events. The ADHD framework supports these counters (Section 5.5), but the level of detail they provide is not required in this context.

The micro-benchmarks presented in this chapter have a very narrow focus and a high control of their behaviour; they are first and foremost designed to measure the performance impact of different inefficiencies. Hardware performance counters provide little extra information in such a controlled environment and this chapter will therefore not consider the hardware performance counter features of the benchmark framework.

6.2 COMPUTATIONS

For computations, the definitions of latency, throughput and bandwidth remain unaltered. Section 3.1.3 presents an extensive discussion on how processor architectures may be optimised to increase their instruction throughput and hide latencies.

Furthermore, the Operation Latency (OL) of a computational instruction is constant, i. e. once an instruction's operands are available, the instruction will take a fixed amount of cycles to complete as all its dependencies have been resolved. This dissertation will therefore focus on the benchmarking of *memory* instructions and transfers, the design of which is discussed by the following chapter.

6.3 MEMORY

Caches employ different techniques to optimise memory read and write operations. This may cause different performance characteristics depending on the balance between reads and writes in a program. This warrants distinct definitions for reading and writing latencies.

- *read latency*: the number of cycles between a read request and the moment when the requested data becomes available in a General-Purpose Register (GPR).
- *write latency*: the number of cycles between a write request and the moment when the data to be written becomes available to subsequent instructions.

- *throughput*: the ratio of satisfied memory requests to processor cycles.
- *bandwidth*: the maximal ratio of the requested amount of data to processor cycles.

Memory bandwidth is a function of throughput and request size. Achievable bandwidth is ultimately limited by the largest data type supported by memory instructions. A throughput benchmark using the largest scalar data type available will therefore also indicate maximal bandwidth for scalar code; analogous for vectorised code.

6.3.1 *Effective Micro-Benchmark Design*

Latency benchmarks must strive to eliminate latency hiding opportunities and thus limit Instruction-Level Parallelism (ILP) as much as possible. Chapters 3 and 4 provide extensive information on processor architectures and how they exploit ILP. This section introduces different observable performance characteristics in a processor, which parameters may impact their behaviour, and how a micro-benchmark may be devised to isolate a single performance characteristic, or even observe the impact of individual micro-architectural features.

6.3.1.1 *Read Latency*

The input parameter for memory read instructions is the memory address on which they should operate. Their output is the value corresponding to the memory address. A dependency may therefore be introduced by making the memory address location dependent on the result of a previous memory instruction.

This scheme may be generalised by ensuring that the sequence of visited memory addresses is cyclic. This eliminates the need for boundary checks and allows the effective benchmark to perform an unlimited amount of reads. This enables a benchmark to generate a contiguous stream of memory instructions regardless of the size of the memory and the memory access pattern that are being tested. The ability to execute nothing but memory instructions eliminates potential complications by reducing complexity and therefore increases the accuracy and reliability of the benchmark.

This benchmark estimates effective memory latency by measuring the amount of cycles that are needed to perform a predetermined amount of read requests. The amount of requests must be sufficient such that the impact of external random events is minimised.

6.3.1.2 *Write Latency*

The input parameters for memory write instructions are the memory address on which they should operate and the value they should store. They do not have a canonical output value, i.e. their result is a side-effect in memory, and all values associated with the instruction are known at the time it requires its input values. It is therefore not possible to generate a sequence of write instructions whose input is dependent on their *output* as such. Their side-effect does incur different kinds of dependencies, i.e. the three hazards of modifying data: Read After Write (RAW), Write After Read (WAR) and Write After Write (WAW).

A sequence of memory write instructions where each instruction requires the value that has been written by the preceding instruction may therefore not be reordered and features no *ILP*. However, pipeline forwarding will completely bypass the memory subsystems and no true latency may be observed. Furthermore, the value to be written has to be constant, as it is always both input and output of each instruction and memory write instruction can not modify the value itself. Optimising compilers will eliminate such code and emit a single write instruction with the final value or propagate the constant to a global value, or both.

Classic bandwidth benchmarks such as *bandwidth* [Smi] are essentially a manually coded loop of nothing but memory instructions. This technique pushes the hardware to its limits, but it is not suitable to characterise *cache* behaviour of optimised applications. Cache memories are too small to store the amount of data needed for statistically significant tests, which forces benchmarks to be constructed in a way such that they write multiple times to the same location. Overwriting data which has never been accessed makes the original memory operation redundant¹. Such redundant operations are often removed by optimising compilers, making this technique unsuitable for micro-benchmarks implemented in an application language.

In summary, observing raw memory latencies for write requests is not feasible because they have no canonical output value. No read requests are allowed and the produced value will therefore always be truncated by subsequent writes. This is not the case when the memory address varies, but two writes to different addresses are functionally independent and thus also invalidate the latency results. Bandwidth may however still be observed as explained in detail in Section 6.3.1.5.

6.3.1.3 Throughput: Latency Hiding

As discussed in Section 6.3.1.2, it is not feasible to measure a baseline for raw write latencies. However, *maximal* throughput *can* be achieved (Section 6.3.1.4). Memory write performance is therefore best characterised by observing the impact of inefficiencies on bandwidth. The remainder of this section considers read request latency hiding only.

The memory request throughput inherent to a single program sequence is a measure for the latency hiding capabilities of the hardware. *ILP* may be introduced to the chasing pointers benchmark by simply chasing multiple pointers. The program may then request the next value for pointer_b before the request for pointer_a is completed. Care should be taken to interleave the requests for different pointers, as the dependencies would otherwise persist through program order.

The *ADHD* framework (Chapter 5) supports the running of benchmarks with multiple threads. Multiple instances of the chasing pointers benchmark may either share or duplicate the array of offsets. As no data is being modified during the effective benchmark, there is no benefit in duplication and therefore no reason to render the implementation more complex. Multi-threaded chasing pointers benchmarks will increase the request rate to shared memory subsystems.

¹Memory-mapped I/O is the exception, as repeated writes may have a significant side-effect.

6.3.1.4 Read Bandwidth

Elementary reduction operations are well-suited for memory read bandwidth operations as they accumulate a single result value. The linear memory traversal provides the hardware with a best-case scenario as it allows for maximal effectiveness of memory prefetching by the hardware. A minimal operation is required to generate semantically sound code that is representative of genuine applications. Calculating the sum of all elements in a set is a reduction operation in its simplest form and represents the maximal performance for bandwidth-bound reduction implementations in general.

6.3.1.5 Write Bandwidth

Peak memory write bandwidth is best approached using streaming stores, as they reduce overhead by avoiding memory reads. A benchmark may initialise a contiguous memory segment with a constant value to provide the hardware with a best-case scenario.

6.3.2 Generalisation & Parameterisation

The chasing pointer benchmark can be maximally generalised by parameterising the following properties:

- *Array length* determines which memory levels in the hierarchy will be included in the observation.
- *Array alignment* determines how data will be stored in caches and page tables, as the caching location of data is based on its full physical address.
- The *access pattern* determines the degree in which subsequent memory accesses are predictable, and therefore in which degree a processor may recognise and optimise for it. Linear memory accesses, both increasing and decreasing through the memory address space are highly predictable and should result in the best possible performance. The random memory access patterns are less predictable, but may in turn be configured to represent different access patterns, and might even be tuned to simulate the access patterns present in genuine applications.
- The *datum size* will determine how efficiently the processor may interpret or convert different data types to that of a memory address. The best possible performance will be observed when this parameter equals the machine word size, i. e. 64 bit on the vast majority of contemporary hardware.
- The number of *threads* and their processor *affinity* will observe how multiple cores interact. This parameter alone is likely to influence many possible observations and the interaction of multiple threads often significantly increases the entropy in the system.
- The amount of *Instruction-Level Parallelism (ILP)* determines the potential for latency hiding which will increase request throughput.

- The *vectorisation* of code will result in more data to be requested at once, i. e. it will help maximise the size of memory transfers.

A straightforward implementation could store a (cyclic) memory address sequence where each value is the next address to read from, but such a direct approach has two main disadvantages: it does not reflect any behaviour of genuine applications and it is limited to a single data type, i. e. memory addresses, which is platform-dependent.

The alternative is to devise a scheme using indirect addressing, i. e. use a base address and relative offsets. On some machines this might be problematic, as additional calculations are introduced. However, address generation is a separate (early) stage in the instruction pipeline in many processor architectures, including Intel Architecture 32-bit (IA₃₂), Intel® 64 and Knights Corner. Such machines may store the invariant base address in a register and load offsets from memory, which are subsequently forwarded to the Address Generation Unit (AGU).

This section will limit its scope to the latter implementation method as the practical scope of this thesis is limited to architectures featuring a separate address generation stage.

```
// configuration
unsigned num_iterations = 1000;
unsigned array[] = {1, 2, 3, 4, 0};

// execution
unsigned index = 0;
while (num_iterations--)
    index = array[index];
```

Listing 6.2: Chasing Pointers: Principle

Listing 6.2 illustrates the core of the chasing pointer algorithm. The values of an array are initialised to the array offset of their respective successors. An initial offset is initialised, and the effective benchmark repeatedly generates new values for the offset by indexing the array using its current value.

```
// configuration
unsigned length = 1024;
unsigned array[length];

unsigned index;
bool increasing = true;

if (increasing) {
    for (index = 0; index < length - 1; ++index)
        array[index] = index + 1;
    array[index] = 0;
}
else {
    array[0] = length - 1;
    for (index = 1; index < length; ++index)
        array[index] = index - 1;
}
```


Listing 6.3: Chasing Pointers: Setup for Linear Access

The size of the array will determine which memory levels are characterised. Small arrays will generate memory access patterns with high spatial and temporal locality which results in extremely high cache-hit rates. Higher levels in the memory hierarchy are accessed as the array size increases. Listing 6.3 illustrates how length may be parametrised and how simple ascending or descending memory access patterns may be encoded in the array. Note that the number of read accesses to the array must be equal or greater than its length as the effective memory pattern may span a smaller memory region than intended otherwise.

Observed latency will become an average of the different latencies of all memory subsystems involved as the array size increases. An important effect of prefetching caches is that they greatly reduce observed latency for main memory. Sufficiently unpredictable memory access patterns negate the benefit of prefetching caches, as speculative loads become impossible and cache-line hit rates reduce.

```
// (configuration as shown in Listing '\ref{lst:chasinglin}')
// initialisation step: encode index as values
for (unsigned index = 0; index < length; ++index)
    array[index] = index;

// shuffle the array
// assume random(min, max) randomly generates a value in the
// range ]min,max[
unsigned index, swap_pos, swap_val;
for (index = 0; idx < length - 1; ++index) {
    swap_pos = random(index, length);
    swap_val = array[index];
    array[index] = array[swap_pos];
    array[swap_pos] = swap_val;
}
```

Listing 6.4: Chasing Pointers: Setup for Random Access

Listing 6.4 illustrates how a random memory access pattern can be encoded in the array. The algorithm is a minor variation on the *Fisher-Yates* shuffle known as *Satollo's algorithm* [Sat86]. It generates an array which encodes a *complete* cycle. It is paramount for the cycle to equal the length of the array, as some memory locations may be left unread otherwise. This would cause inaccurate results as the memory range subject to the test could be smaller than requested. In-depth studies on more intricate properties of Satollo's algorithm are available in literature, e.g. Helmut Prodingen presents a detailed mathematical analysis on two parameters, "the number of moves" and "distance", in [Pro02].

The number of linear permutations of a set of n distinct elements is $P_n = n!$. For fixed circular permutations, all elements are arranged on a *fixed* circle, i.e. the circle may not be lifted from the plane and turned over. All cyclic permutations of the elements become equivalent as the circle may be rotated. The number of cyclic permutations of a linear sequence equals its length; the number of circular permuta-

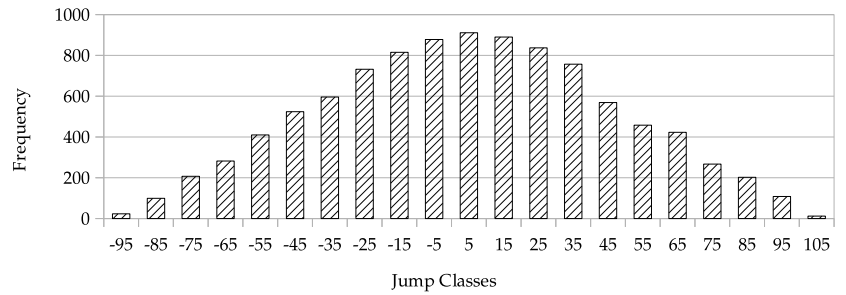
tions of a set of n distinct elements is therefore $P_n = \frac{n!}{n} = (n-1)!$. In free circular permutations, the circle of elements *may* be turned over, making predecessor and successor relations equivalent, thereby halving the total number of possible permutations: $P_n = \frac{(n-1)!}{2}$.

Given a uniformly distributed random number generator, the implementation of *Satollo's* algorithm will generate all permutations of a set with equal probability. Distinct permutations only exist for sets containing at least four (distinct) elements as shown in Equation 6.1. However, the number of possible permutations grows in a factorial order of magnitude and just eleven elements already result in almost two million possible permutations: $\frac{(11-1)!}{2} = 1814400$.

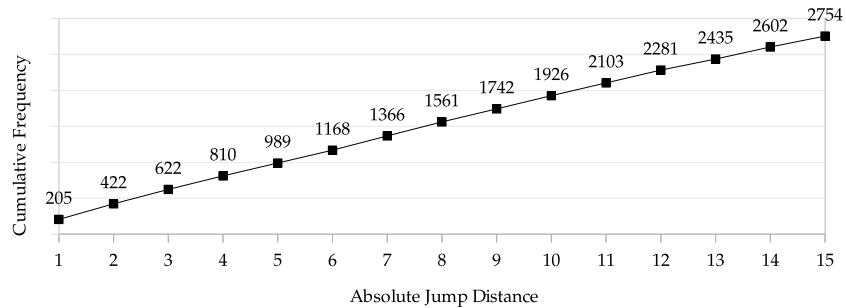
$$\frac{(n-1)!}{2} > 1 \Leftrightarrow (x-1)! > 2! \Leftrightarrow x-1 > 2 \Leftrightarrow x > 3 \quad (6.1)$$

The vast amount of possibilities and the absence of bias make it extremely unlikely to generate a non-chaotic (i. e. predictable) memory access pattern. Optimisations in the memory architecture are equally unlikely to be effective and the benchmark will be exposed to raw, unoptimised memory access latencies. Additionally, by running the same micro-benchmark set-up multiple times using a different random number sequence, accidental bias will be practically impossible.

Figure 6.1 shows a histogram of the jump distances generated by the algorithm, with a bin size of ten. 10 000 data points were generated by running the algorithm 100 times for an array size of 100 and a uniformly distributed Random Number Generator (RNG).



(a)



(b)

Figure 6.1: random access pattern with uniform RNG

The histogram shows that 9.11 % of the jump distances fall in the interval $]-5, 5]$. Assume a cache line size of 64 B and datum size of 8 B, i. e. 8 values per line. Any two consecutive requests that access data within a distance of 4 elements have an average chance of 68.75 % to hit the same cache line ($P_{\text{hit}} = \frac{l \bmod d}{l}$). Note this is a low estimate as the distribution suggests that smaller jumps are more frequent. Nevertheless, it is reasonable to assume that less than 9 % of any two consecutive memory requests will hit the same cache line.

The effect of the random number generator distribution on the generated access pattern is profound. Users may require different properties for the access pattern to approach different real-world access patterns or to emphasise the effect of certain optimisations in the hardware architecture. Figure 6.2 illustrates the effect of using a binomial distribution with $P_n = 0.5$ for the random number generator. Figure 6.3 illustrates the effect when $P_n = 0.1$ and $P_n = 0.9$.

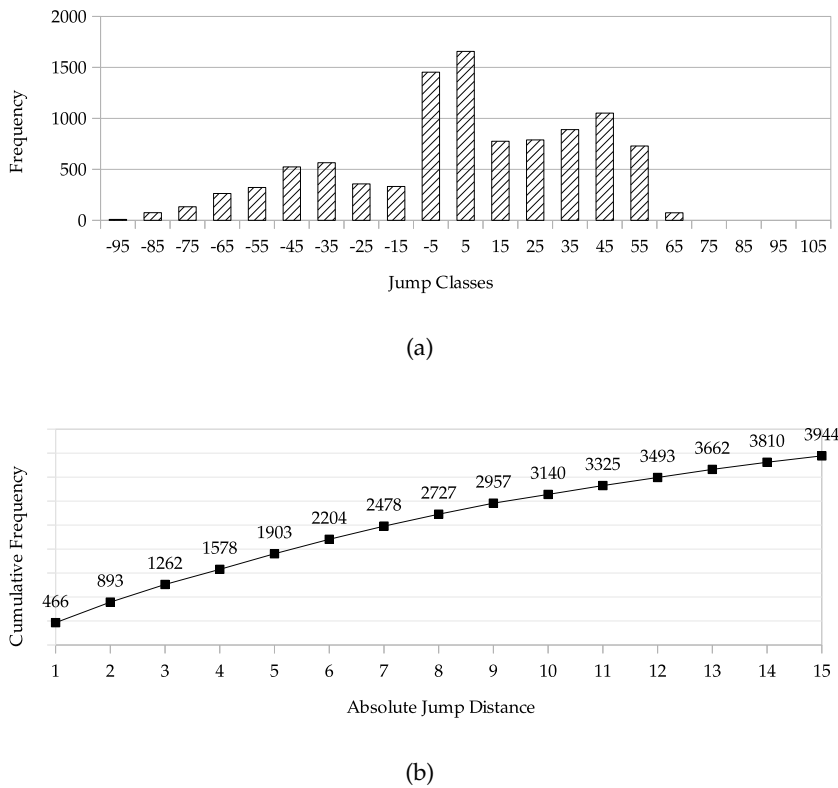


Figure 6.2: random access pattern; symmetrical binomial RNG ($P_n = 0.5$)

6.3.3 Throughput

Two main approaches can be taken to implement an increase in ILP for the chasing pointers benchmark. A first is illustrated in the listing below. The main advantage of this approach is that the level of ILP in the benchmark is merely a function of a run-time parameter. However, it requires that an optimising compiler successfully determines that all different values of the array are in fact independent.

```
// (configuration as shown in Listing 6.2)
```

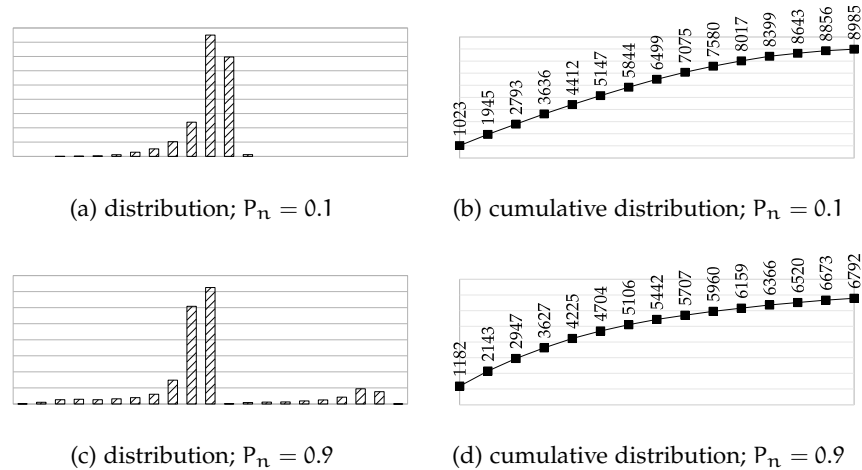


Figure 6.3: random access pattern; asymmetrical binomial RNG

```
// different starting positions to prevent register recycling
unsigned num_indices = 4;
unsigned index_set[num_indices] = {0, 1, 2, 3};

while (num_iterations--)
    for (unsigned i = 0; i < num_indices; ++i)
        index_set[i] = array[index_set[i]];

```

Listing 6.5: Chasing Pointers: Exploiting ILP — arrays

An alternative approach is the same in concept, but is implemented using separate distinct local variables. Less restrictions apply to such an implementation regarding possible optimisations, and a compiler is therefore more likely to generate maximally efficient code. The clear downside of the approach is the manual duplication of the variables.

Note that this problem can not be solved using C++ template programming, as they can not produce new *identifiers*, i.e. variable names, and therefore no varying amount of variables in a function body. The process can be semi-automated by some repetitive preprocessor macro code, which is withheld for the sake of the reader's sanity. Interested readers may consult the ADHD source code, available online as specified in Chapter 5.

```
// (configuration as shown in Listing 6.2)
// different starting positions to prevent register recycling
unsigned index_a = 0; unsigned index_b = 1;
unsigned index_c = 2; unsigned index_d = 3;

while (num_iterations--) {
    index_a = array[index_a]; index_b = array[index_b];
    index_c = array[index_c]; index_d = array[index_d];
}

```

Listing 6.6: Chasing Pointers: Exploiting ILP — local variables

Either approach is best implemented as a standalone function that eventually returns the sum of all indexes, in order to force the compiler to compute all values. Otherwise, it may opt to omit all code that does not contribute to the return value as it does not have a semantically significant side-effect.

One might opt to qualify the different variables as *volatile*, which essentially informs the compiler that the value contained in this variable may change outside the scope of the program. This is however an ill-advised practice as changing the code in such manner renders it less representative of genuine applications that do not qualify their variables.

6.3.4 Bandwidth

A peak bandwidth benchmark must provide a best case scenario for the hardware and ensure they are not bound by any subsystem other than memory itself.

Feature	Request Size	Request Rate
arithmetic intensity	✓	✓
contiguous data access		✓
packed data	✓	
aligned data		✓
streaming writes		✓
multiple threads		✓

Table 6.1: Memory bandwidth design & optimisation

Arithmetic intensity, i. e. the ratio of computational instructions to memory instructions, should be minimal. Benchmarks for theoretical peak bandwidth must additionally ensure they are not bound by *processor* instruction throughput and should therefore scale to multiple processors. Reading may be optimised more aggressively than writing. These requirements lead to three different forms of relevant memory bandwidth benchmarks:

BENCHMARK PERFORMANCE & EFFICIENCY ANALYSIS

In this chapter we apply the benchmarks developed in Chapter 6 to a Xeon® E5-2690 v2 Central Processing Unit (CPU) and the Xeon Phi™. The former yields a number of baseline results that serve as a reference for the benchmark accuracy and the results for the Xeon Phi™.

The study on the Xeon Phi™ is much more extensive. The micro-benchmarks are employed to examine the hardware in increasing amounts of detail. A number of interesting and peculiar characteristics of the Xeon Phi™ core performance will be observed.

The baseline tests are performed on a Xeon® E5-2690 v2, which is a 10-core, hyper-threaded, 6-wide superscalar processor. Two of the six execution ports are able to process load and store requests. The first level and second level caches are local to a single core and therefore shared by two hyper-threads. The third and last level cache is shared across the core, i.e. by 20 (hyper-)threads in total. The number of independent instructions to access the cache is therefore not only a function of the number of independent pointers in the benchmark, but also of the number of threads. All threads of a multi-threaded chasing pointer benchmark will start reading from the same memory location, i.e. they have inter-thread temporal locality and are likely to benefit from shared lower-level caches if they run synchronously.

The compiler used for the benchmarks in this section is *icpc* (ICC) 14.0.1 20131008. The baseline benchmarks have a limited scope as their goal is only to generate essential results that can be used as a reference.

7.1 XEON® E5-2690 V2: BASELINE RESULTS

7.1.1 Chasing Pointers

7.1.1.1 L1 Cache

The L1 cache is 32 kB and its latency depends on how memory is addressed, as explained by the following extract from the *Intel® Architectures Optimisation Reference Manual* [Cor12]:

The common load latency is five cycles. When using a simple addressing mode, base plus offset that is smaller than 2048, the load latency can be four cycles. This technique is especially useful for pointer-chasing code.

Table 7.1 shows the configuration parameters of the L1 cache benchmark. The results are shown in Figure 7.1.

The remaining sections discuss the different causes for the behaviour of the benchmark and the hardware. Section 7.1.1.2 discusses the observed latency for a single independent instruction stream. Section 7.1.1.3 addresses the strong performance penalty for 14 or more independent instruction streams. Section 7.1.1.4 addresses the earlier onset of efficiency declension for multiple threads and the impact of compiler optimisations as corollary.

A systematic approach if incremental micro-benchmarking helps identifying many specific hardware performance characteristics and the impact of inefficiencies.

trials	idx	length	align	seq	threads	streams	read
8	8 B	16 kB	4 kB	U^a	[1, 2]	[1, 16]	256 MB

^a U : cyclic permutation obtained with a uniformly distributed RNG

Table 7.1: Xeon® E5-2690 v2 L1 cache: chasing pointers configuration

7.1.1.2 Simple Addressing

As previously discussed, the minimal L1 cache load latency for the Xeon® E5-2690 v2 may be either 4 or 5 cycles, depending on the address generation result. The single-threaded no-Instruction-Level Parallelism (ILP) benchmark instance shows a heightened relative standard deviation at 6.27%. This being a relatively low value by itself notwithstanding, it does indicate that the instance invokes different behaviour in the execution core.

The raw results shown in Table 7.2 suggest that the majority of address calculations (i.e. $base + offset$) were able to benefit from the optimised access latency of 4 cycles, with the exception of the first trial. It is therefore reasonable to assume an unfortunate memory allocation prevented the optimisation in the first trial.

trial	1	2	3	4	5	6	7	8
cycles	5.00	4.22	4.22	4.21	4.20	4.20	4.20	4.20

Table 7.2: Xeon® E5-2690 v2L1 cache: raw data for single thread, no ILP

7.1.1.3 Register Spilling

Different pointers may be chased without overhead as long as enough registers are available to store the complete state of the benchmark. Listing 7.1 shows the x86-64 machine code for an implementation of the routine shown in Listing 6.6, using 13 independent instruction streams.

From a high-level point of view, the `inc` instruction increments the loop counter, the `cmp` instruction tests the loop condition, and the `jb` instruction continues the loop when the end condition is not met by jumping to the beginning of the code listing. The remaining instructions are all `mov` instructions which store data in a location determined by their last argument. This listing confirms that the compiler was able to use as much registers as needed, as none are written to more than once in a single iteration.

```

408f13:  inc    %rdx
408f16:  mov    (%rax,%rbx,8),%rbx
408f1a:  mov    (%rax,%rbp,8),%rbp
408f1e:  mov    (%rax,%r9,8),%r9
408f22:  mov    (%rax,%rdi,8),%rdi
408f26:  mov    (%rax,%r8,8),%r8

```

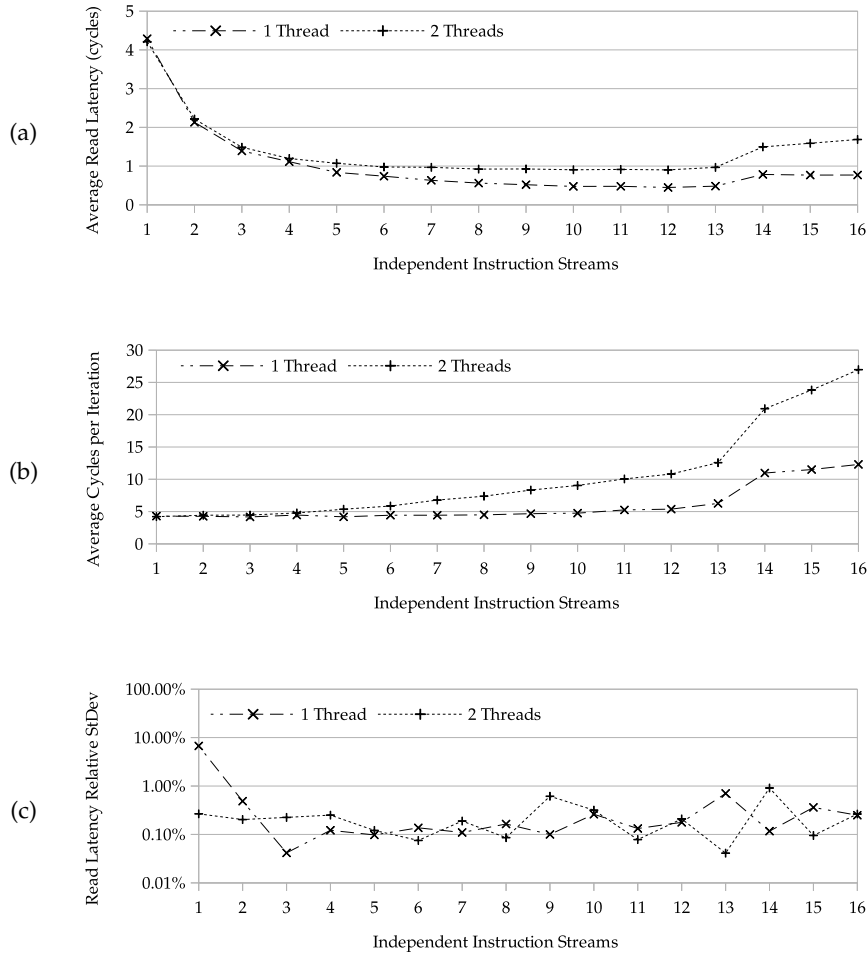


Figure 7.1: Xeon E5-2690 v2: L1 cache latencies

```

408f2a:  mov    (%rax,%r15,8),%r15
408f2e:  mov    (%rax,%rsi,8),%rsi
408f32:  mov    (%rax,%r10,8),%r10
408f36:  mov    (%rax,%r12,8),%r12
408f3a:  mov    (%rax,%r11,8),%r11
408f3e:  mov    (%rax,%r14,8),%r14
408f42:  mov    (%rax,%r13,8),%r13
408f46:  mov    (%rax,%rcx,8),%rcx
408f4a:  cmp    0x18(%rsp),%rdx
408f4f:  jb     408f13

```

Listing 7.1: ILP 13: no register spilling

If not enough registers are available, the compiler will have to generate code to temporarily store the state of one or more pointers to memory in order to free a register. Listing 7.2 shows the same routine as before, but using 14 independent instruction streams. The generated code supports the hypothesis that register spilling is the cause of diminished performance, as two registers are written to multiple times. These extra operations are caused by limitations in the hard-

ware which requires additional bookkeeping code to be emitted by the compiler. The additional instructions and non-register memory accesses result in extra cycles spent in every iteration of the loop, and cause an increase in the observed L1 cache latency.

```

408d10:    mov    0x20(%rsp),%rax    # write %rax
408d15:    inc     %rdx
408d18:    mov    0x30(%rsp),%rsi    # write %rsi
408d1d:    mov    (%rax,%rbp,8),%rbp
408d21:    mov    (%rax,%rsi,8),%rsi  # write %rsi
408d25:    mov    %rsi,0x30(%rsp)
408d2a:    mov    0x28(%rsp),%rsi    # write %rsi
408d2f:    mov    (%rax,%rbx,8),%rbx
408d33:    mov    (%rax,%r10,8),%r10
408d37:    mov    (%rax,%rdi,8),%rdi
408d3b:    mov    (%rax,%r9,8),%r9
408d3f:    mov    (%rax,%r8,8),%r8
408d43:    mov    (%rax,%r15,8),%r15
408d47:    mov    (%rax,%r11,8),%r11
408d4b:    mov    (%rax,%r13,8),%r13
408d4f:    mov    (%rax,%r12,8),%r12
408d53:    mov    (%rax,%r14,8),%r14
408d57:    mov    (%rax,%rcx,8),%rcx
408d5b:    mov    (%rax,%rsi,8),%rax  # write %rax
408d5f:    mov    %rax,0x28(%rsp)
408d64:    cmp    0x18(%rsp),%rdx
408d69:    jb     408d10

```

Listing 7.2: ILP 14: register spilling

7.1.1.4 Automatic Loop Unrolling

The best case latency for L1 cache requests is 4 cycles and the lowest observed latency for a single request is 4.21 cycles (Table 7.2). The overhead caused by the looping infrastructure is therefore completely hidden as the minimal number of cycles per loop must be at least 5 cycles otherwise. This effect may be attributed to either the execution hardware or compiler optimisations, or both. The impact of most compiler optimisations can be determined by selectively enabling and disabling those of interest.

Few optimisations are possible for a chasing pointer benchmark, as all memory instructions are strictly dependent. The loop infrastructure dependencies are more flexible however: it is a bound loop, i. e. the number of iterations is known when the loop is entered. A compiler may duplicate the code in the loop body multiple times, which reduces the number of loop iterations required by the same factor. More resources can now be dedicated to relevant computations as the iteration variable is less often incremented and tested for the end condition.

The optimisation is easily observed in machine code. Listing 7.3 and Listing 7.4 respectively show the unoptimised and unrolled versions.

```

409ddf:    inc     %rdx

```

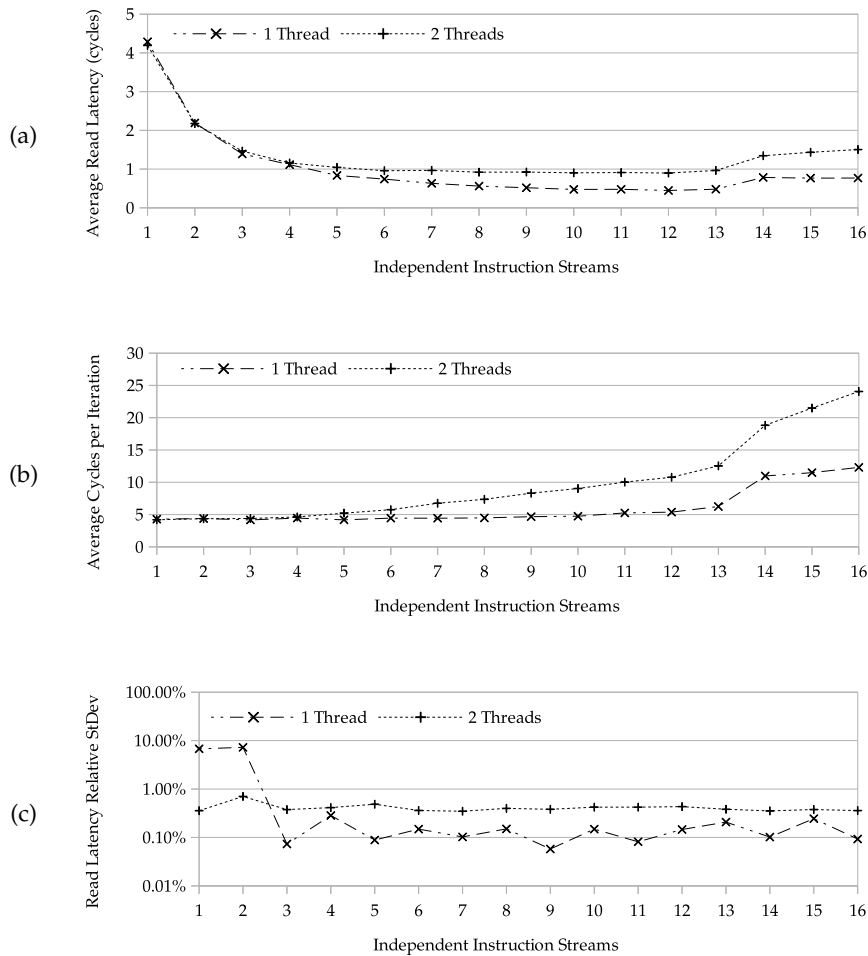



Figure 7.2: Xeon E5-2690 v2: L1 cache latencies - no loop unrolling

```

409de2:  mov    (%r11,%rsi,8),%rsi
409de6:  cmp    %r8,%rdx
409de9:  jb     409ddf

```

Listing 7.3: no ILP: no loop unroll

```

40a508:  mov    (%r11,%r9,8),%rax
40a50c:  inc    %rsi
40a50f:  mov    (%r11,%rax,8),%r9
40a513:  cmp    %rdx,%rsi
40a516:  jb     40a508

```

Listing 7.4: no ILP: automatic loop unroll

The compiler unrolled the core benchmark loop by a factor of two, i. e. the loop body length is doubled and the number of iterations is halved. The first mov instruction is moved to the top to execute it as soon as possible to improve latency hiding.

However, loop unrolling is not always beneficial and may have a negative impact on global performance when multiple threads compete for shared resources. The results of running a benchmark with loop unrolling disabled and an otherwise unmodified configuration (see Table 7.1) are shown in Figure 7.2. There are no significant differences in performance for the single-threaded benchmark, but when two threads access the L1 cache, the average latency is lowered. Not unrolling loops results in a richer instruction mix and memory requests are spread more evenly in machine code. This has an amortising effect on the temporal distribution of requests issued to the cache which in turn enhanced the memory controller occupancy and reduces the number of possible conflicts.

7.1.1.5 L2 Cache

The L2 cache is 256 kB and its theoretical latency is 12 cycles. Table 7.3 shows the configuration parameters of the L2 cache benchmark. The results are shown in Figure 7.3.

trials	idx	length	align	seq	threads	streams	read
8	8 B	240 kB	4 kB	U^a	[1, 2]	[1, 16]	256 MB

^a U : cyclic permutation obtained with a uniformly distributed RNG

Table 7.3: Xeon[®] E5-2690 v2 L2 cache: chasing pointers configuration

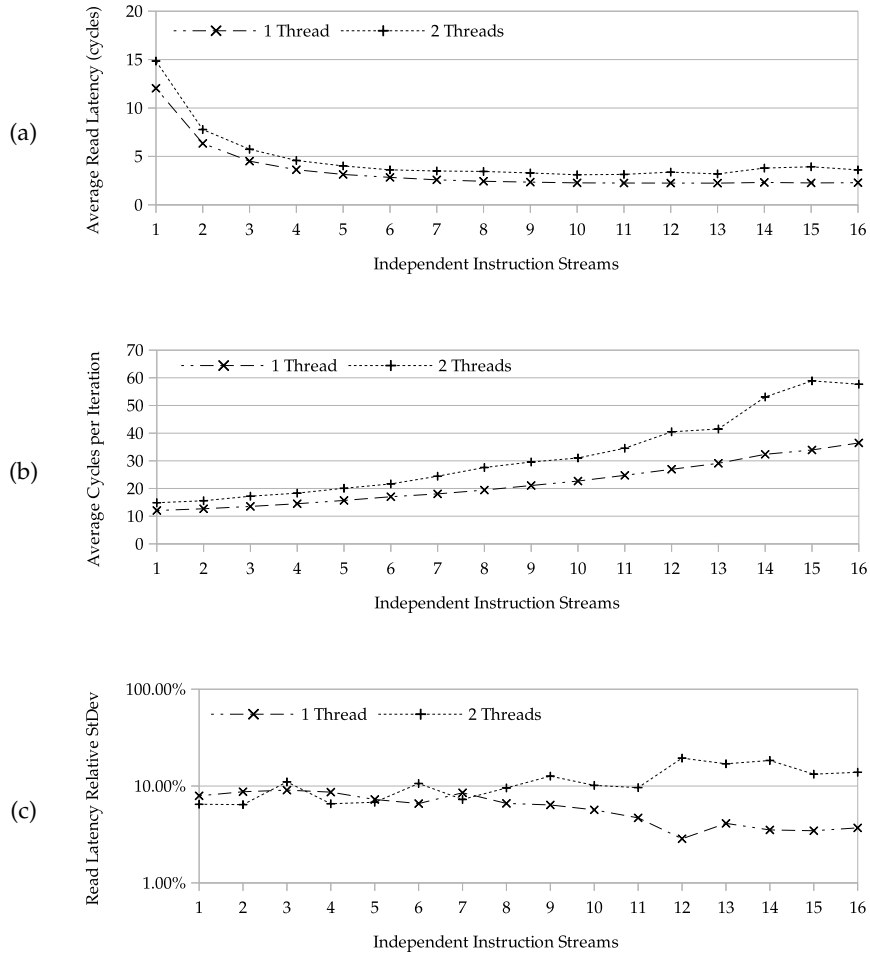


Figure 7.3: Xeon E5-2690 v2: L2 cache latencies

7.1.1.6 L3 Cache

The third and last level cache of the Xeon® E5-2690 v2 is shared across all 10 cores, i. e. 20 hyper-threads. The L3 cache is 25 MB and its theoretical latency is 26-31 cycles. Table 7.4 shows the configuration parameters of the L3 cache benchmark. The results are shown in Figure 7.4.

trials	idx	length	align	seq	threads	streams	read
8	8 B	24 MB	4 kB	U^a	[1, 2]	[1, 16]	256 MB

^a U : cyclic permutation obtained with a uniformly distributed RNG

Table 7.4: Xeon® E5-2690 v2 L3 cache: chasing pointers configuration

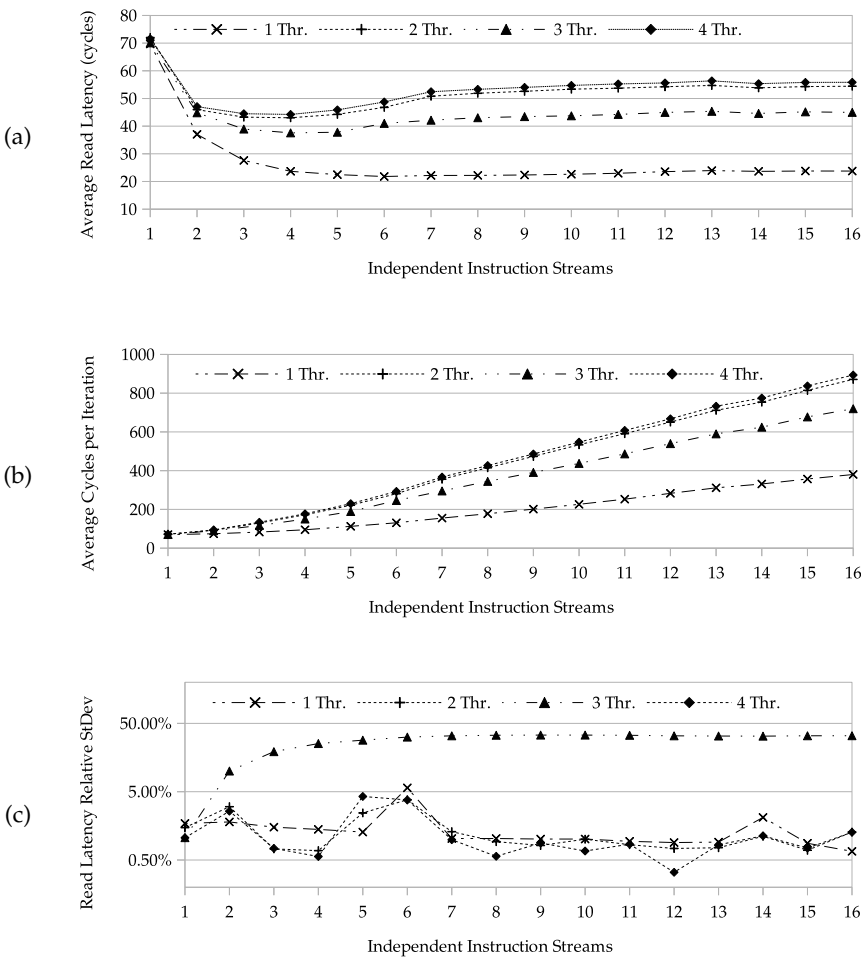


Figure 7.4: Xeon E5-2690 v2: L3 cache latencies

7.1.1.7 Main Memory

The third and last level cache of the Xeon[®] E5-2690 v2 is shared across all 10 cores, i. e. 20 hyper-threads. The L3 cache is 25 MB and its theoretical latency is 26-31 cycles. Table 7.5 shows the configuration parameters of the Random-Access Memory (RAM) cache benchmark. The results are shown in Figure 7.5.

trials	idx	length	align	seq	threads	streams	read
8	8 B	240 MB	4 kB	U^a	[1, 2]	[1, 16]	512 MB

^a U : cyclic permutation obtained with a uniformly distributed RNG

Table 7.5: Xeon[®] E5-2690 v2 RAM: chasing pointers configuration

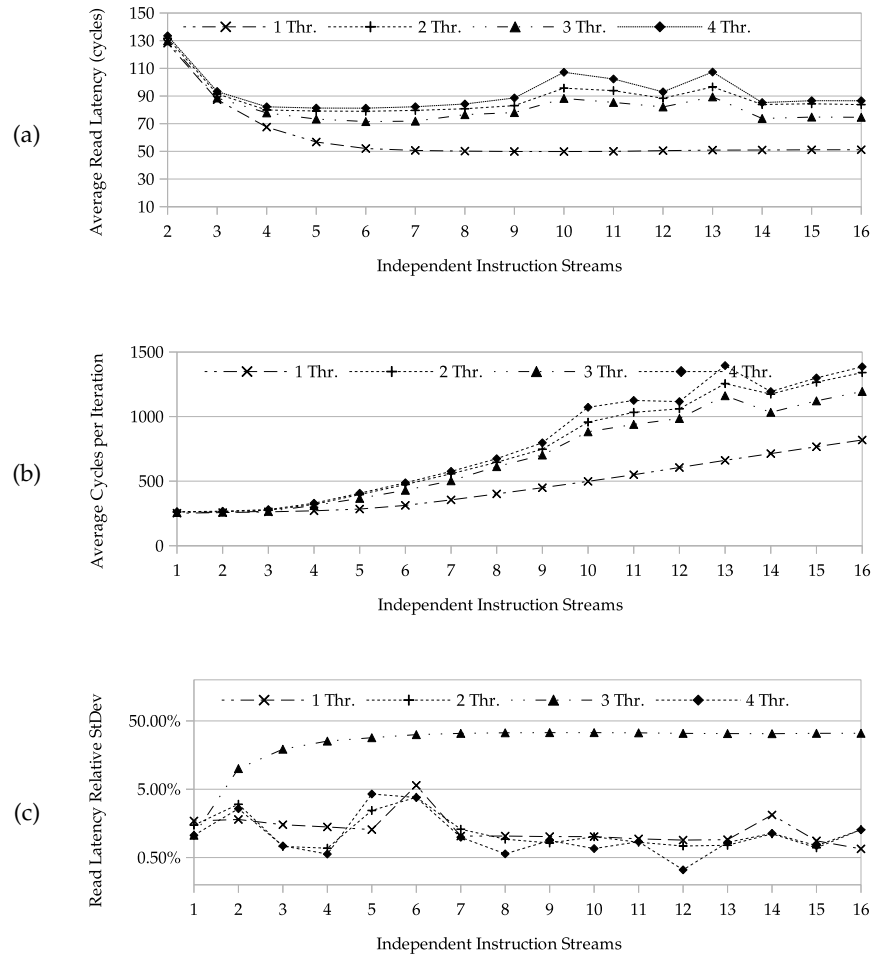


Figure 7.5: Xeon E5-2690 v2: RAM latencies

7.2 XEON PHI™: DETAILED ANALYSIS & CONTEXTS

The detailed analysis of the Knights Corner (KNC) follows a systematic approach. A first benchmark helps identifying interesting or unexpected behaviour in the memory systems of the KNC. Additional benchmarks may then further explore the inexplicable behavioural patterns and indicate their cause with high precision.

7.2.1 Memory Hierarchy Overview

Figure 7.6 (page 73) shows a full overview of observed memory latencies across the whole system. The latency values are averaged across 8 trials. The variance between the trials is presented as the *relative* standard deviation, i.e. $\frac{\sigma}{\bar{x}}$, which directly reflects the significance of the variance. No single experiment exceeds a relative standard deviation of 0.5%, which clearly proves that the data is highly consistent.

7.2.1.1 First Impressions

Access to main memory is predictably many times slower (both axis scale linearly) than the smaller cache memories, and show almost no variance in the test results. The machine does seem to prefer certain memory sizes more than others, which is likely to be correlated with memory page boundaries causing Translation Lookaside Buffer (TLB) misses. More focused micro-benchmarks may identify the cause with more precision.

A good insight in how the caching systems behave is required for proper main memory analysis, as the caches can not be completely disabled. Furthermore, a good understanding of the cache allows one to assess its impact and degree of importance to obtain peak performance in relation to main memory.

7.2.1.2 Cache Hierarchy: Random Access

Figure 7.7 (page 74) zooms in on the full overview of Figure 7.6 and shows more detail about the latencies of the cache subsystems. The increased amount of detail reveals many peculiar properties of the cache. According to the hardware specifications in Table 3.7, there are two levels of cache: 32 kB of L1 cache and 512 kB of L2 cache. Interpreting the results is best done by breaking down the graph in different parts:

- 1 kB → 32 kB
The first-level cache has a clear, low and constant observable latency. However, despite the practically constant *average* observed latency, the relative noise is at an increased level compared to the range from 96 kB to 192 kB. This indicates that external factors may influence the results, which may range from simply imprecise measurements to intricate instruction pipeline behaviour.
- 33 kB → 112 kB
The increased level of noise at the boundary between the two first cache levels can simply be attributed to the properties of the memory access randomisation algorithm: different trials

may experience a different amount of hits in the first-level cache, which reduces the observed latency.

- 113 kB \rightarrow 220 kB
The observed latency slowly increases as the ratio of first-level to second-level cache hits drops. This is expected and normal behaviour.

- 221 kB \rightarrow 280 kB
The noise in the results is at its highest in this range, which clearly indicates that an unknown variable influences the measurements between trials. It appears around a clear corner point in the data, which suggests that the random memory accesses in this range have some random chance to fall on either side of this point. Additionally, this corner point is situated around 256 kB, which is half the size of the second-level cache.

The specifications in Table 3.7 state that the second-level cache is divided in two banks. It is therefore highly likely that the corner point is caused by *banking conflicts*, i. e. memory requests to the same memory bank are being serialised.

- 280 kB \rightarrow 430 kB
The probability of a first-level cache hit in function of the tested memory size can be approached by the inverse function ($f(x) = x^{-1} = 1/x$): $P(L1_{hit}) = \frac{|L1|}{|m|}$. The probability of a second-level cache hit in function of the tested memory size can therefore be approached by the negative inverse function: $P(L2_{hit}) = 1 - P(L1_{hit}) = -P(L1_{hit}) + 1$. The expected behaviour would therefore be that the observed latency converges to the upper-bound of the second-level cache.

The observed behaviour does not correspond with what is expected: the latency continues to increase linearly in function of memory size. This supports the previous hypothesis that the corner point is caused by banking conflicts in the second-level cache as the number of potential conflicts increases linearly in function of the size of the memory range.

- Beyond 430 kB, it is difficult to assess what may be the likely cause for the additional and severe performance penalty based on these results alone. Further investigation to the simpler cases will likely show shared causes and allow one to eliminate them, thereby reducing the total number of potential causes.



Figure 7.6: KNC full memory latency overview

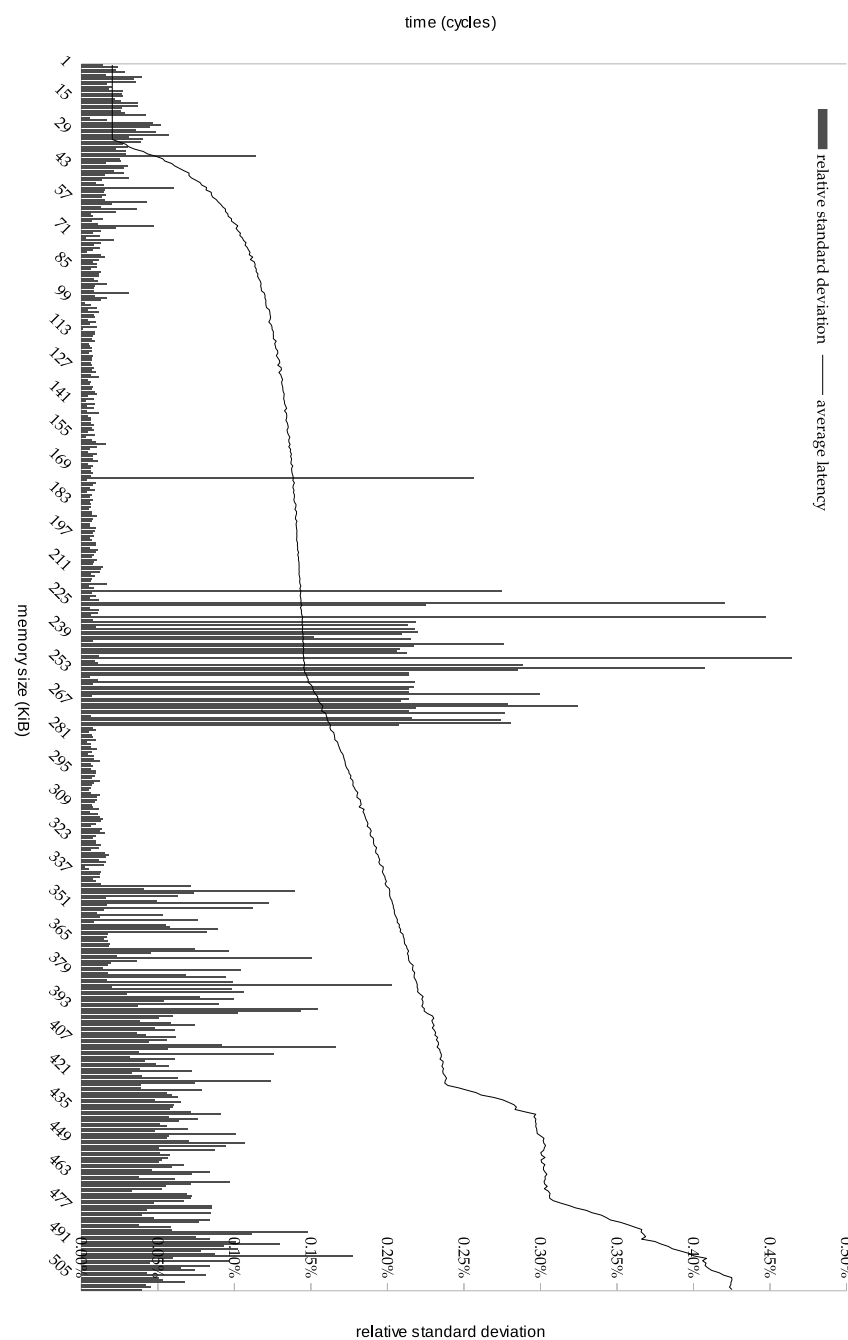


Figure 7.7: KNC cache hierarchy latency detail

7.2.1.3 Cache Hierarchy: Linear Access

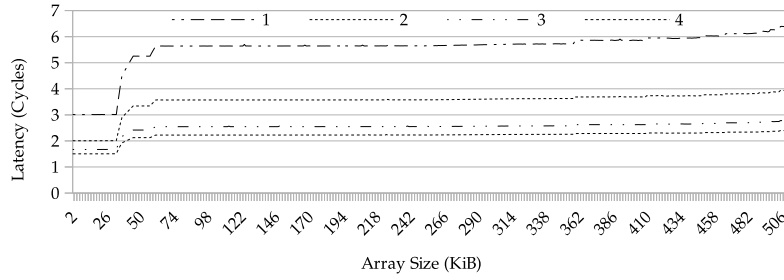


Figure 7.8: KNC linear cache accesses

Figure 7.8 shows the effect of accessing the cache subsystems with a linear memory access pattern and different levels of ILP. The graph clearly shows that the penalties observed in the overviews in Figures 7.6 and 7.7 can be attributed to the *random* memory access pattern. Additionally, it clearly shows that adding additional instructions effectively starts hiding the latencies of the memory instructions caused by the Address Generation Interlock (AGI).

7.2.2 Disabled Loop Unrolling & Address Generation Interlock

The results of the linear memory access pattern suggests that the core is, notwithstanding that the benchmark is single-threaded, able to obtain an observed latency of less than 2 cycles. This clearly shows that the core scheduling mechanisms are more complex than suggested by the Xeon Phi™ system programmer’s manual (see Chapter 3) and that the core is able to issue more than one instruction every two cycles and that it is therefore not limited by the fact that the thread is only selected for execution every other cycle.

no unroll: ILP 1

To better understand what is happening, we disable compiler optimisations and disassemble the code generated by the Intel® compiler for the chasing pointer algorithm. For the version with an ILP of only 1, the code is as follows:

```

40: 48 ff c2      inc    %rdx
43: 49 8b 34 f3    mov    (%r11,%rsi,8),%rsi
47: 49 3b d0      cmp    %r8,%rdx
4a: 72 f4        jb     40

```

Listing 7.5: no unroll: ILP 1

The `inc` instruction increments the loop counter, the `mov` instruction is the actual instruction in which we are interested, and the `cmp` and `jb` instructions respectively check the loop counter and repeat the loop body if needed.

If we model its execution using the computational pipelines of the pipeline model, we can see that this version should require 4 cycles per iteration, as illustrated by Table 7.6

1	2	3	4	...
inc	mov ₁	cmp	jb	→

Table 7.6: no unroll, [ILP 1](#): 4 loop cycles

no unroll: ILP 2

For further comparison, we analyse the generated code for the version with an [ILP](#) of 2 and the same configurations otherwise. The resulting code follows:

```

4c:  48 ff c2      inc    %rdx
4f:  4f 8b 1c de    mov    (%r14,%r11,8),%r11
53:  4f 8b 14 d6    mov    (%r14,%r10,8),%r10
57:  48 3b d6      cmp    %rsi,%rdx
5a:  72 f0      jb     4c

```

Listing 7.6: no unroll: [ILP 2](#)

Here we see equivalent code, but with two memory instructions. It would therefore be reasonable to assume an execution time of 5 cycles per iteration. However, if we effectively run this code (not shown) we observe a memory latency of 3 cycles per instruction, i. e. 6 cycles per loop.

To explain this behaviour, we first observe that the Xeon Phi™ is a 64 bit machine, but that the memory instructions are only 32 bit each. Therefore we synthesise that the PF stage fetches a word, i. e. 64 bit for the program instruction stream, and places it in the intermediate buffer, which will make effectively *two* memory instruction available for further processing.

Additionally, we observe that, as stated by the Xeon Phi™ programmer's manual, a jump instruction triggers an instruction cache miss, which as a consequence empties the intermediate buffer. Therefore, when the jump instruction occurs on an *odd* cycle when the core is running only one thread, the PPF will have switched to the dummy thread, and only in the next cycle will the following machine word be fetched of the instruction stream.

If we model this behaviour according to the pipeline model, we obtain the simulation shown in Table [7.7](#)

1	2	3	4	5	6	
inc	mov ₁	mov ₂	cmp	jb	PF stall	→

Table 7.7: no unroll, [ILP 2](#): 6 loop cycles

no unroll: ILP 14

The number of available registers is typically relatively limited in [CPU](#) architectures, and the Xeon Phi™ is no different. Not illustrated in this chapter are the many experiments with many intermediate values for [ILP](#) to identify when *register spilling* occurs.

However, during these tests, we observed that the performance of the benchmark could not accurately be predicted by the pipeline mod-

elwhen **ILP** was set to 14. The generated code is illustrated below, and teaches us that as much as 6 additional memory instructions are introduced.

d0:	48 8b 44 24 20	mov	0x20(%rsp),%rax
d5:	48 ff c2	inc	%rdx
d8:	48 8b 74 24 28	mov	0x28(%rsp),%rsi
dd:	48 8b 1c d8	mov	(%rax,%rbx,8),%rbx
e1:	48 8b 34 f0	mov	(%rax,%rsi,8),%rsi
e5:	4e 8b 0c c8	mov	(%rax,%r9,8),%r9
e9:	48 89 74 24 28	mov	%rsi,0x28(%rsp)
ee:	48 8b 74 24 30	mov	0x30(%rsp),%rsi
f3:	48 8b 34 f0	mov	(%rax,%rsi,8),%rsi
f7:	48 8b 2c e8	mov	(%rax,%rbp,8),%rbp
fb:	4e 8b 04 c0	mov	(%rax,%r8,8),%r8
ff:	48 89 74 24 30	mov	%rsi,0x30(%rsp)
# snipped 8 regular, independent mov instructions			
124:	48 8b 44 24 18	mov	0x18(%rsp),%rax
129:	48 3b d0	cmp	%rax,%rdx
12c:	72 a2	jb	d0

Listing 7.7: no unroll: **ILP** 14

Two important observations can be made: if an **ILP** of 14 introduces 3 additional stores and 3 additional loads, then must an **ILP** of 11 be optimal. Secondly, the total amount of cycles required by the loop should be 14 (**ILP**) + 6 (spilling) + 3 (loop overhead) = 23. However, when we run this benchmark we can infer that the total latency of the loop is actually 26 cycles in practice.

If we model the assembly code by the pipeline model, and we label the mov instructions that cater for register spilling mov_α and mov_β , we obtain the execution profile shown in Table 7.8.

1	2	3	4	5	6-7	8	9	
$\text{mov}_{\alpha l}$	inc	$\text{mov}_{\beta l}$	mov_1	AGI	mov_{2-3}	$\text{mov}_{\beta s}$	$\text{mov}_{\beta l}$...
...	10-11	12-14	15	16-23	24	25	26	
...	AGI	mov_{4-6}	$\text{mov}_{\beta s}$	mov_{7-14}	$\text{mov}_{\alpha s}$	cmp	jmp	→

Table 7.8: no unroll, **ILP** 14: 26 loop cycles

The simulation clearly shows that the Intel® compiler failed to generate an optimal ordering for the memory instructions, which results in the loss of cycles 5, 10 and 11. This is caused by the AGI of the Xeon Phi™ pipeline, which dictates that when the result of an instruction is required in the address generation phase by its direct successor, a delay of at least three cycles will have to be inserted.

More specifically, $\text{mov}_{\beta l}$ stores a value to the %rsi register, which in the instruction stream is required only two cycles later. The same problem reoccurs when a spilling load produces a value that is directly required by the subsequent instruction. In total, this causes 3 lost cycles in a loop that would otherwise require 23 cycles, i.e. a 13 % increase in execution time.

7.2.3 Enabled Loop Unrolling & Register Spilling

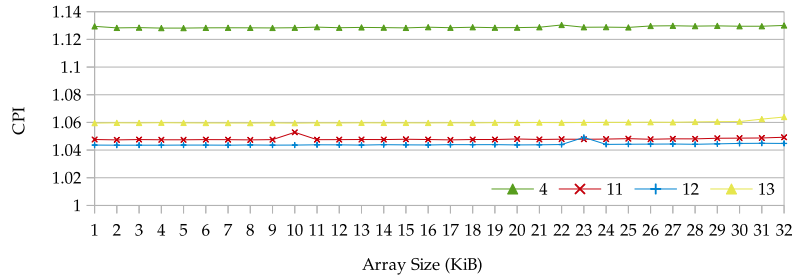


Figure 7.9: KNC linear cache accesses

In the previous subsection, we have learned that the execution of a tight loop in a single thread will always require an even number of cycles on the Xeon Phi™ core. This section focuses on the benefits of loop unrolling, which will reduce the relative cost of this effect as well as the relative cost of the loop overhead.

Figure 7.9 illustrates that the issue latency for dependent micro-instructions can successfully be hidden by providing a sufficiently large *ILP*. Remarkably, while using an *ILP* of 13 results in register spilling, the performance penalty is almost indistinguishable. The code that is generated for this version is illustrated in the listing below, and shows that a total of 18 instructions has been generated, but without the introduction of an *AGI* cost. The additional memory instructions do not require more than a single cycle because of the single-cycle latency of the first-level cache.

```

c0: 48 8b 74 24 20    mov    0x20(%rsp),%rsi
c5: 48 ff c0          inc    %rax
c8: 48 8b 1c de        mov    (%rsi,%rbx,8),%rbx
cc: 48 8b 2c ee        mov    (%rsi,%rbp,8),%rbp
d0: 4e 8b 0c ce        mov    (%rsi,%r9,8),%r9
d4: 4e 8b 04 c6        mov    (%rsi,%r8,8),%r8
d8: 48 8b 14 d6        mov    (%rsi,%rdx,8),%rdx
dc: 48 8b 3c fe        mov    (%rsi,%rdi,8),%rdi
e0: 4e 8b 3c fe        mov    (%rsi,%r15,8),%r15
e4: 4e 8b 14 d6        mov    (%rsi,%r10,8),%r10
e8: 4e 8b 24 e6        mov    (%rsi,%r12,8),%r12
ec: 4e 8b 1c de        mov    (%rsi,%r11,8),%r11
f0: 4e 8b 34 f6        mov    (%rsi,%r14,8),%r14
f4: 4e 8b 2c ee        mov    (%rsi,%r13,8),%r13
f8: 48 8b 0c ce        mov    (%rsi,%rcx,8),%rcx
fc: 48 8b 74 24 18    mov    0x18(%rsp),%rsi
101: 48 3b c6           cmp    %rsi,%rax
104: 72 ba             jb     c0

```

Listing 7.8: *ILP* 13: register spilling

7.2.4 Alignment for L2 Random Access

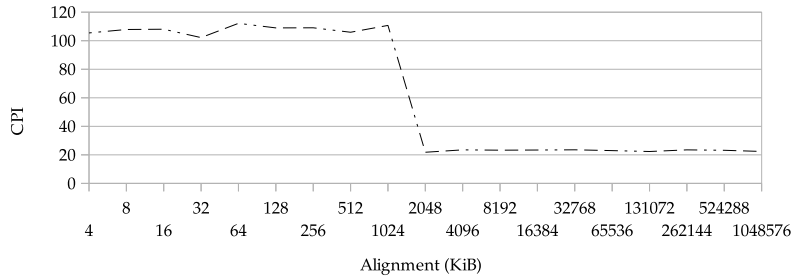


Figure 7.10: KNC random L2 cache access: varying alignment

The search for the cause of the abysmal performance of the second-level cache has eluded not only the author, but even contacts with Intel® were not able to pinpoint the cause. The graph shown in Figure 7.10 shows the second-level cache performance for varying degrees of alignment of the data set.

It is the result of a very exhaustive search for possible causes, for which the Analyze Diverse Hardware Demeanors (ADHD) framework proved to be a very valuable tool. Based on earlier findings, we configured the benchmark with an ILP of 11, as this is the highest amount of read instructions that is possible without causing register spilling, and which results in an even total amount of instructions in the payload loop. We selected a data size of 500 kB to approach the full size of the second-level cache without risking an overflow to main memory.

We chose to start the alignment at 4 kB as this corresponds with the memory page size and benefits optimally from the TLB infrastructure in that regard. However, it is clear that components other than the TLB causes the performance penalties. A likely explanation is that a very strong alignment helps in creating a ‘perfect’ mapping of the data to the second-level cache.

SUMMARY

In this section we have shown the following through the use of micro-benchmarks:

- The Xeon Phi™ scheduling mechanics cause tight loops to always take an even amount of cycles, when only one thread is executed on the core.
- Second-level cache latency and performance for random access patterns is heavily dependent on the memory alignment of the data set.
- The hardware-specified latencies of the first-level cache, i. e. one cycle, can be obtained with only a single thread. This is an effective improvement on the state of the art [Fan+14].
- The first-level cache latency of 1 cycle allows the core to perform register spilling operations with no cost other than the execution of the memory instructions themselves, which is also a single cycle.

- Random memory access for the second-level cache is approximately 20 cycles when the data set is sufficiently aligned.

CONCLUSIONS

In this dissertation, we observed that the world of High-Performance Computing (HPC) is evolving rapidly. Supercomputers will soon enter the exascale era and will be able to perform more than a quintillion (10^{18}) operations per second. However, where the hardware is evolving rapidly, software fails to keep up. Software efficiency is in danger of becoming a major problem to employ the vast computational power hardware has to offer.

Xeon Phi™

We selected the Intel® Xeon Phi™ co-processor for our case study and in-depth analysis. The Intel® Many Integrated Core (MIC) architecture is a key component in the company's commitment to reach exascale performance by 2018. The Xeon Phi™ co-processor code-named Knights Corner (KNC) is the most recent MIC implementation and provides the main compute power of the fastest supercomputer in the world, the Tianhe-2 (TH-2). To develop a thorough understanding of the MIC architecture, we first performed a theoretical in-depth study on general compute architectures.

The Pipeline Model

We selected the Graphics Processing Unit (GPU)-based pipeline model to model the highly complex nature of processor execution cores. The pipeline model is not finished and still work in progress of the Department of Electronics and Informatics (ETRO) lab of the Vrije Universiteit Brussel (VUB). It nevertheless proved to be sufficiently malleable to adapt to the Central Processing Unit (CPU)-based MIC core architecture and required only a slight reinterpretation.

Micro-benchmarks that create a controlled environment to observe the latency involved with different operations can obtain the parameters for the pipeline model. The contexts of the pipeline model require the empirical exploration of the hardware, which is possible by sweeping many parameters of different benchmarks.

ADHD

The amount of micro-benchmarks that are required may vary by hardware platform and the desired precision of the model. This dissertation presented ADHD, a software framework that greatly facilitates creating highly configurable and portable benchmarks. It trivially supports parameter sweeping which greatly helps in characterising hardware behaviour and therefore the investigation for contexts. We opted to implement the framework in C++11 to increase its relevance to genuine applications and to leverage many of the abstraction features of the programming language.

Micro-Benchmarks

Different micro-benchmarks have been developed, which have helped understanding the intricate details of the operation of MIC processor core. They have also shown that the Intel[®] programming manual does not present an entirely accurate description of the scheduling mechanisms in the KNC core. They have illustrated that the cache behaviour of the Xeon Phi[™] is very atypical compared to modern CPU architectures and that the Intel[®] compiler does not always generate optimal code from fairly straightforward C++ code.

Pipeline Model Parameters & Contexts

The benchmarks have been able to generate necessary hardware parameters for the pipeline model. A major result in this respect is the analysis of the complex behaviour and latencies of memory instructions *and transfers*, which corrects what is currently assumed in literature, e.g. in [Fan+14].

This dissertation was able to identify multiple latencies and contexts for the Xeon Phi[™]:

- Memory instructions have an issue latency (λ) of 1 cycle. Their completion latency (Λ) is 3 cycles when the subsequent instruction requires its result for address generation and 1 cycle otherwise.
- Adding multiple memory instructions relaxes the data dependency relations and increases the available Instruction-Level Parallelism (ILP) even on the in-order Xeon Phi[™] core architecture. This technique is able to effectively hide the Λ of memory instructions, when appropriately arranged in the assembly code. The compiler plays an important role in this regard and forms a static pipeline model context.
- The random memory access latency for the second-level cache is subject to many different factors, which may quadruple (or even eight-fold in some worst-case scenarios) the observed latencies even when all accessed data fits in the cache. Optimal latency for random access was observed as approximately 17 cycles. The nominal bad-case observed latencies counted more than 80 cycles. The access pattern and memory alignment were the dominant factors and therefore define a context of the pipeline model.
- The latency benchmarks have also shown that the peculiar instruction scheduling mechanism of the Xeon Phi[™] may introduce an additional (dead) cycle to small loops, even when sufficient instructions are available to fill the instruction pipeline. This observation was helpful in understanding the core operation, but we chose not to define this artefact as a context as its impact is only significant in extremely small, special-case loops.

8.1 FUTURE WORK

The current Xeon Phi[™] architecture is code-named KNC and its core architecture is based on the in-order Intel[®] p54c architecture which was first introduced in 1994. The successor to KNC will be code-named

Knights Landing (KNL) and will employ radically different designs. It will replace the current inter-core communication *ring* with a *mesh* network, which will radically change the communication characteristics. An even more significant change is the new core architecture it will deploy: Out-of-Order Execution (OOE) cores based on the new low-power and highly power-efficient *silvermont* architecture, which will also be featured in the Intel® Atom® product line.

ADHD

One of the main design goals for the Analyze Diverse Hardware De-meanors (ADHD) framework is portability to different hardware architectures. This has already been partly illustrated by the baseline tests on a Xeon® CPU. The framework and its benchmark may analyse the new architecture using the exact same code, which will illustrate the changes in performance characteristics of the new hardware.

Further development will help in approaching its long-term design goal of becoming a feasible platform for prototyping highly efficient genuine applications. Additional benchmarks and configurations will help in generating more complete hardware characterisations. General improvement to the end-user interface will turn ADHD in a feasible alternative for micro-benchmarking by the general public.

OpenCL

The ADHD framework currently only supports threaded benchmarks using pthreads. With the addition of Open Computing Language (OpenCL) [SGS10] support, it could also be employed to characterise the performance and efficiency of GPUs as well. Additionally, it would allow to compare the performance of a C++ implementation versus an OpenCL implementation of the same benchmark.

Pipeline Model

The work in this dissertation has generated new insights on how coarse-grained hardware may also be fitted by the pipeline model and opened new directions for its ongoing research. A simulator may be developed to calculate the estimated efficiency and run-time of larger applications, according to the parameters and semantics of the pipeline model. In combination with continued development of ADHD, performance and efficiency estimations for the next-generation Xeon Phi™ using the pipeline model may be feasible.

BIBLIOGRAPHY

PRINTED REFERENCES

- [BF96] Dick Buttlar and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing*. " O'Reilly Media, Inc.", 1996.
- [Bro+89] Jim Browne et al. "1988 Gordon Bell Prize". In: *IEEE Softw.* 6.3 (May 1989), pp. 78–85. ISSN: 0740-7459. DOI: [10.1109/52.28127](#).
- [DH13] Jack Dongarra and Michael A Heroux. "Toward a new metric for ranking high performance computing systems". In: *Sandia Report, SAND2013-4744* 312 (2013).
- [Eis+09] M. Eisenbach et al. "A Scalable Method for Ab Initio Computation of Free Energies in Nanoscale Systems". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. SC '09. Portland, Oregon: ACM, 2009, 64:1–64:8. ISBN: 978-1-60558-744-8. DOI: [10.1145/1654059.1654125](#).
- [Fan+14] Jianbin Fang et al. "Test-driving Intel Xeon Phi". In: *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering*. ICPE '14. Dublin, Ireland: ACM, 2014, pp. 137–148. ISBN: 978-1-4503-2733-6. DOI: [10.1145/2568088.2576799](#).
- [HK09] Sunpyo Hong and Hyesoon Kim. "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness". In: *ACM SIGARCH Computer Architecture News*. Vol. 37. 3. ACM. 2009, pp. 152–163.
- [HP11] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011. ISBN: 012383872X, 9780123838728.
- [Juc+03] Guido Juckeland et al. "BenchIT-Performance Measurements and Comparison for Scientific Applications." In: *PARCO*. 2003, pp. 501–508.
- [JW89] Norman P Jouppi and David W Wall. *Available instruction-level parallelism for superscalar and superpipelined machines*. Vol. 17. 2. ACM, 1989.
- [Knu74] Donald E. Knuth. "Computer Programming as an Art". In: *Communications of the ACM* 17.12 (Dec. 1974), pp. 667–673.

- [Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89683-4.
- [M+96] Larry W McVoy, Carl Staelin et al. "Imbench: Portable Tools for Performance Analysis." In: *USENIX annual technical conference*. San Diego, CA, USA. 1996, pp. 279–294.
- [Mir+99] A. A. Mirin et al. "Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System". In: *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*. Supercomputing '99. Portland, Oregon, USA: ACM, 1999. ISBN: 1-58113-091-0. DOI: [10.1145/331532.331601](#).
- [Moo98] G.E. Moore. "Cramming More Components Onto Integrated Circuits". In: *Proceedings of the IEEE* 86.1 (Jan. 1998), pp. 82–85. ISSN: 0018-9219. DOI: [10.1109/JPROC.1998.658762](#).
- [MP87] Stephen W Melvin and Yale N Patt. "A Clarification of the Dynamic/Static Interface". In: *Proceedings, 20th Hawaii International Conference on System Sciences*. 1987, pp. 6–9.
- [PP84] Mark S Papamarcos and Janak H Patel. "A low-overhead coherence solution for multiprocessors with private cache memories". In: *ACM SIGARCH Computer Architecture News*. Vol. 12. 3. ACM. 1984, pp. 348–354.
- [Proo2] Helmut Prodinger. "On the analysis of an algorithm to generate a random cyclic permutation". In: *Ars Combinatoria* 65 (2002), pp. 75–78.
- [RH13] S. Ramos and T. Hoefler. "Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi". In: *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. New York City, NY, USA: ACM, June 2013, pp. 97–108. ISBN: 978-1-4503-1910-2.
- [Ros+13] Diego Rossinelli et al. "11 PFLOP/s Simulations of Cloud Cavitation Collapse". In: *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '13. Denver, Colorado: ACM, 2013, 3:1–3:13. ISBN: 978-1-4503-2378-9. DOI: [10.1145/2503210.2504565](#).
- [Sat86] Sandra Sattolo. "An algorithm to generate a random cyclic permutation". In: *Information processing letters* 22.6 (1986), pp. 315–317.
- [SB13] Yakun Sophia Shao and David Brooks. "Energy characterization and instruction-level energy model of Intel's Xeon

- Phi processor". In: *Low Power Electronics and Design (IS-LPED), 2013 IEEE International Symposium on*. IEEE. 2013, pp. 389–394.
- [SGS10] John E Stone, David Gohara and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems". In: *Computing in science & engineering* 12.3 (2010), p. 66.
- [Sim+12] Jaewoong Sim et al. "A performance analysis framework for identifying potential benefits in GPGPU applications". In: *ACM SIGPLAN Notices*. Vol. 47. 8. ACM. 2012, pp. 11–22.
- [SL05] John Paul Shen and Mikko H. Lipasti. *Modern processor design : fundamentals of superscalar processors*. Index. Boston: McGraw-Hill Higher Education, 2005. ISBN: 0-07-057064-7. URL: <http://opac.inria.fr/record=b1129703>.
- [War11] James Warnock. "Circuit Design Challenges at the 14Nm Technology Node". In: *Proceedings of the 48th Design Automation Conference*. DAC '11. San Diego, California: ACM, 2011, pp. 464–467. ISBN: 978-1-4503-0636-2. DOI: [10.1145/2024724.2024833](https://doi.org/10.1145/2024724.2024833).

ONLINE REFERENCES

- [CM13] PAPI Team & Contributors and Phil Mucci. *Performance Application Programming Interface*. 3rd Dec. 2013. URL: <http://icl.cs.utk.edu/papi/> (visited on 23/05/2014).
- [Com] Association for Computing Machinery. *Gordon Bell Prize*. URL: <http://awards.acm.org/bell/> (visited on 01/04/2014).
- [Cor12] Intel® Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Apr. 2012. URL: <http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf> (visited on 03/06/2014).
- [Cor14] Intel® Corporation. *Intel® Xeon Phi™ System Software Developers Guide*. Mar. 2014. URL: <http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf> (visited on 16/05/2014).
- [Don13] Jack Dongarra. *Visit to the National University for Defense Technology Changsha, China*. 3rd June 2013. URL: <http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf> (visited on 28/03/2014).
- [Fen+] Wu-chun Feng et al. *The Green500 List*. URL: <http://green500.org> (visited on 22/04/2014).

- [Gao01] Xiaofeng Gao. *Effective Moore's Laws in High Performance Computing Based on Gordon Bell Prize Winners*. University of California at San Diego. 2001. URL: <http://cseweb.ucsd.edu/~carter/260/bellprize.pdf> (visited on 01/04/2014).
- [HD] Rajeeb Hazra and Boyd Davis. *Technical Computing. Discover Your Parallel Universe*. Supercomputing Conference. URL: http://newsroom.intel.com/servlet/JiveServlet/download/38-28204/SC'13_Intel_presentation.pdf (visited on 14/04/2013).
- [Lab12] Oak Ridge National Laboratory. *ORNL Debuts Titan Supercomputer. Supercomputer combines gaming and traditional computing technologies to provide unprecedented power for research*. 2012. URL: https://www.olcf.ornl.gov/wp-content/themes/olcf/titan/Titan_Debuts.pdf (visited on 10/04/2014).
- [NVI14] NVIDIA. *NVIDIA GeForce GTX 750 Ti. Featuring First-Generation Maxwell GPU Technology, Designed for Extreme Performance per Watt*. 2014. URL: <http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce-GTX-750-Ti-Whitepaper.pdf> (visited on 14/04/2014).
- [Pet+12] Antoine Petit et al. *HPL 2.1. A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. 26th Oct. 2012. URL: <http://www.netlib.org/benchmark/hpl/> (visited on 27/03/2014).
- [Rah13] Reza Rahman. *Intel® Xeon Phi™ Core Micro-architecture. Intel® Xeon Phi™ Cores*. May 2013. URL: <https://software.intel.com/sites/default/files/article/393195/intel-xeon-phi-core-micro-architecture.pdf> (visited on 16/05/2014).
- [Ska11] Kirk Skaugen. *Intel: Accelerating the Path to Exascale*. International Supercomputing Conference. 2011. URL: http://newsroom.intel.com/servlet/JiveServlet/download/2152-20-5259/Intel_ISC_2011_MIC_Kirk_Skaugen_Presentation.pdf (visited on 08/04/2014).
- [Smi] Zack Smith. *Bandwidth: a memory bandwidth benchmark. for x86 / x86_64 based Linux/Windows/MacOSX*. URL: <http://zsmith.co/bandwidth.html> (visited on 16/06/2014).
- [SSF12] Tom Scogland, Balaji Subramaniam and Wu-chun Feng. *The Green500 List: Escapades to Exascale*. International Supercomputing Conference. 2012. URL: <http://www.green500.org/sites/default/files/ISC12-green500.pptx> (visited on 23/04/2014).

- [Str+] Erich Strohmaier et al. *Top 500 Supercomputer Sites*. URL: <http://top500.org> (visited on 27/03/2014).
- [Wel13] Jack Wells. *What does Titan tell us about preparing for exascale supercomputers?* Programme du 32eme Forum ORAP. 10th Oct. 2013. URL: <http://www.irisa.fr/orap/Forums/Forum32/Presentations/Wells.pdf> (visited on 10/04/2014).