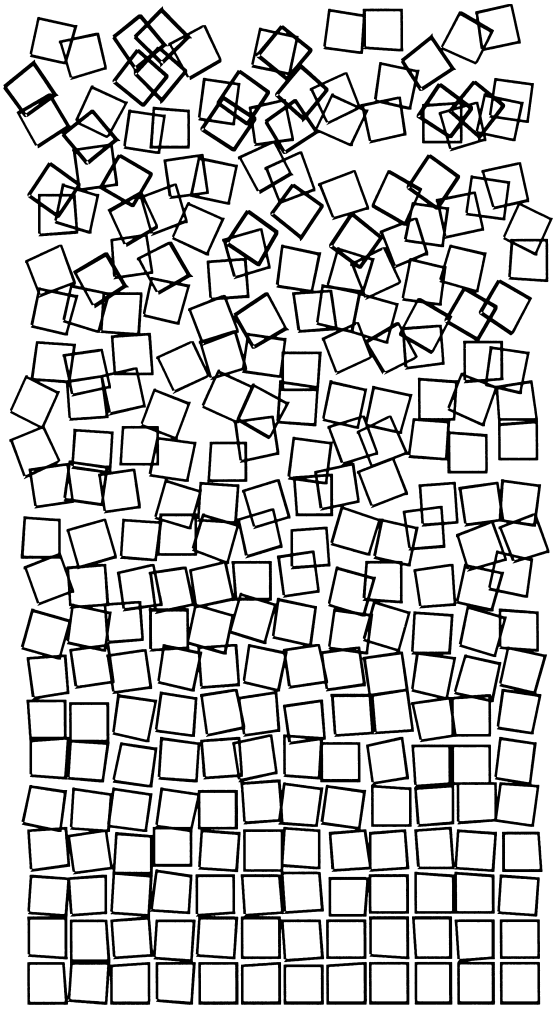
GESELLSCHAFT FÜR INFORMATIK E.V.

PARALLEL-ALGORITHMEN, -RECHNERSTRUKTUREN UND -SYSTEMSOFTWARE

PARS

INFORMATIONSTECHNISCHE GESELLSCHAFT IM VDE



Computergraphik von: Georg Nees, Generative Computergraphik

I n h a l t

Aktuelle PARS-Aktivitäten unter
http://fg-pars.gi.de

ISSN 0177-0454

PARS-Mitteilungen

Gesellschaft für Informatik e.V.,
Parallel-Algorithmen, -Rechnerstrukturen
und -Systemsoftware

Offizielle bibliographische Bezeichnung bei Zitaten:
Mitteilungen - Gesellschaft für Informatik e. V.,
Parallel-Algorithmen und Rechnerstrukturen, ISSN 0177 - 0454

Die PARS-Mitteilungen erscheinen in der Regel einmal pro Jahr. Sie befassen sich mit allen Aspekten paralleler Algorithmen und deren Implementierung auf Rechenanlagen in Hard- und Software.

Die Beiträge werden nicht redigiert, sie stellen die Meinung des Autors dar. Ihr Erscheinen in diesen Mitteilungen bedeutet keine Einschränkung anderweitiger Publikation.

Die Homepage

http://fg-pars.gi.de/

vermittelt aktuelle Informationen über PARS.

GESELLSCHAFT FÜR INFORMATIK E.V.

PARALLEL-ALGORITHMEN, -RECHNERSTRUKTUREN UND -SYSTEMSOFTWARE

PARS

ITG INFORMATIONSTECHNISCHE GESELLSCHAFT IM VDE

CALL FOR PAPERS

28. PARS - Workshop am 21.-22. März 2019

Technische Universität Berlin

http://fg-pars.gi.de/workshops/pars-workshop-2019/

Ziel des PARS-Workshops ist die Vorstellung wesentlicher Aktivitäten im Arbeitsbereich von PARS und ein damit verbundener Gedankenaustausch. Mögliche Themenbereiche sind:

- **Parallele Algorithmen (Beschreibung, Komplexität, Anwendungen)**
- **Parallele Rechenmodelle und parallele Architekturen**
- **Parallele Programmiersprachen und Bibliotheken**
- **Werkzeuge der Parallelisierung (Compiler, Leistungsanalyse, Auto-Tuner)**
- **Parallele eingebettete Systeme / Cyber-Physical Systems**
- **Software Engineering für parallele und verteilte Systeme**
- **Multicore-, Manycore-, GPGPU-Computing und Heterogene Architekturen**
- **Cluster Computing, Grid Computing, Cloud Computing**
- **Verbindungsstrukturen und Hardwareaspekte (z. B. rekonfigurierbare Systeme)**
- **Zukünftige Technologien und neue Berechnungsparadigma für Architekturen (SoC, PIM, STM, Memristor, DNA-Computing, Quantencomputing)**
- **Parallelverarbeitung im Unterricht (Erfahrungen, E-Learning)**
- **Methoden des parallelen und verteilten Rechnens in den Life Sciences (z.B.Bio-, Medizininformatik)**

Die Sprache des Workshops ist Deutsch und Englisch. Für jeden Beitrag sind maximal 10 Seiten vorgesehen. Die Workshop-Beiträge werden als PARS-Mitteilungen (ISSN 0177-0454) publiziert. Es ist eine Workshopgebühr von ca. 100 € geplant.

Termine:

Einreichungsfrist für Beiträge: 23. Januar 2019
Beiträge im Umfang von 10 Seiten (Format: GI Lecture Notes in Informatics: https://gi.de/service/publikationen/lni/, nicht vor-veröffentlicht) sind in elektronischer Form unter folgendem Link einzureichen: **https://easychair.org/conferences/?conf=pars2019**

Benachrichtigung der Autoren bis **15. Februar 2019**
Druckfertige Ausarbeitungen bis **31. August 2019** (nach dem Workshop)

Programmkomitee: *A. Döring, Zürich • N. Eicker, Jülich • T. Fahringer, Innsbruck • D. Fey, Erlangen • V. Heuveline, Heidelberg • R. Hoffmann, Darmstadt • B. Juurlink, Berlin • W. Karl, Karlsruhe • J. Keller, Hagen • E. Maehle, Lübeck • U. Margull, Ingolstadt • E. W. Mayr, München • J. Mottok, Regensburg • W. E. Nagel, Dresden • M. Philippsen, Erlangen • K. D. Reinartz, Höchstadt • B. Schnor, Potsdam • M. Schulz, München • P. Sobe, Dresden • C. Trinitis, München • T. Ungerer, Augsburg • R. Wanka, Erlangen*

Nachwuchspreis: Der beste Beitrag, der auf einer Diplom-/Masterarbeit oder Dissertation basiert, und von dem Autor/der Autorin selbst vorgetragen wird, wird auf dem Workshop von der Fachgruppe PARS mit einem Preis (dotiert mit 500 €) ausgezeichnet. Co-Autoren sind erlaubt, der Doktorgrad sollte zum Zeitpunkt der Einreichung noch nicht verliehen sein. Die Bewerbung um den Preis erfolgt durch E-Mail an die Organisatoren bei Einreichung des Beitrages.

Veranstalter: GI/ITG-Fachgruppe PARS, **http://fg-pars.gi.de**

Organisation:

Prof. Dr. Ben Juurlink, Technische Universität Berlin, Architektur eingebetteter Systeme, 10587 Berlin, Germany, Tel.: +49-30-314-73130, E-Mail: b.juurlink@tu-berlin.de

Daniel Maier, Technische Universität Berlin, Architektur eingebetteter Systeme, 10587 Berlin, Germany, Tel.: +49-30-314-22286, E-Mail: daniel.maier@tu-berlin.de

Prof. Dr. Wolfgang Karl (PARS-Sprecher), Karlsruher Institut für Technologie, Rechnerarchitektur und Parallelverarbeitung, 76131 Karlsruhe, Germany, Tel.: +49-721-608-43771, E-Mail: karl@kit.edu

28. PARS-Workshop

Seite

(Hrsg.): W. Karl, B. Juurlink
Proceedings 28th PARS Workshop

The Evolution of Secure Hash Algorithms

F. Pfautsch, N. Schubert, C. Orglmeister, M. Gebhart, P. Habermann, B. Juurlink
Technische Universität Berlin, Embedded Systems Architecture, Berlin, Germany
{f.pfautsch, n.schubert, c.orglmeister, gebhart, p.habermann, b.juurlink}@tu-berlin.de

Abstract: Hashing algorithms are a popular tool for saving passwords securely or file verification. Storing plain-text passwords is problematic if the database gets exposed. However it is also a problem if the used hashing algorithm is outdated. Short passwords can be attacked with brute-force search, hence recommendations of a minimal password length are common. Given that computer performance increased significantly during the last decades, outdated hashes, especially generated by short passwords, are vulnerable today. We evaluate the resilience of SHA-1 and SHA-3 hashing against brute-force attacks on a 24-core dual-processor system, as well as on a modern UltraScale+ FPGA. Reaching a peak performance of 4.45 Ghashes/s, we are able to find SHA-1 hashed passwords with a length of up to six characters within three minutes. This time increases by a factor of 5.5× for the more secure SHA-3 algorithm due to its higher complexity. We furthermore present a study how the average cracking times grows with increasing password length. To be resilient against brute force attacks, we therefore recommend a minimum password size of at least 8 characters, which increases the needed computing time to several days (SHA-1) or weeks (SHA-3) on average.

Keywords: Hashing; SHA-1; SHA-3; Keccak; FPGA; vectorization

1 Introduction

In the past, numerous databases with hashed passwords got exposed. Additionally, history has shown that weaknesses of algorithms will be found. Hence the increasing security of hashing algorithms and their standardizations is very important to protect passwords. It is also important to use strong hashing algorithms and passwords of sufficient length. If someone tries to find the matching password to a given hash, a lot of work is required in terms of computing power. As hardware has been getting more performant in recent years, hashing algorithms need to keep up to continue being strong against collision attacks, i. e. generating the same hash with different input data, or brute-force attempts to find matching results for passwords. A secure hashing algorithm is therefore characterized by the time, memory utilization and cost in terms of hardware and energy usage it takes to crack a hash.

We compare two different generations of the Secure Hash Algorithm, namely SHA-1 and SHA-3 (with a hash length of 256 bit), in hardware and software to find out how secure passwords of a certain length are. SHA algorithms are standardized by the US National Institute of Standards and Technology (NIST). SHA-2 and SHA-1 are similar, which is the

reason why we chose not to look into SHA-2 further. The formula $h(x) = y$ represents a general hash equation, where h stands for the hash function and the y for the computed hash. We are interested in finding the input value x which represents the password.

The main component of our comparisons is the performance, i. e. how fast hashes can be cracked. In our work we use plain brute-force for cracking passwords hashed with SHA-1 and SHA-3. The hashes are computed on-the-fly meaning we didn't use precalculated rainbow tables and thus also support salted and peppered hashes. Furthermore, we analyze the performance and the cost factor.

The paper is organized as follows: Section 2 provides a brief historical background and the basic operating principle of the Secure Hash Algorithms family. Subsequently our experimental setup is described in Section 3. In Section 4 we present our findings and evaluate the differences between SHA-1 and SHA-3 in hardware and software. Afterwards, Section 5 draws conclusions of the results.

2 Secure Hash Algorithms

The concept of hashing data is used for different occasions, such as comparing hashes of passwords for logins, verifying error-free data transmission or transaction identification in blockchains. Cryptographic hash functions are characterized by collision-avoiding properties, reasonable speed while still providing security and being able to process input of different lengths. Secure hashing algorithms are becoming more important with the growing performance of hardware.

2.1 SHA-1 and SHA-2

The earlier released SHA-0 was quickly superseded by SHA-1 because of serious vulnerabilities [na95]. SHA-1, first published by the US National Security Agency (NSA) in 1995, produces a message digest ("hash") for a given input of 160 bits. It also improved the already existing message digest algorithms of the MD-family e. g. MD5 and MD4 by Ronald L. Rivest [Ri92] by using a similar construction but generating a larger hash value.

First published in 2001, SHA-2 was meant to be used alongside SHA-1 providing higher security by generating longer hash values and being more complex by calculating two intermediary functions instead of just one. Both, SHA-1 and SHA-2, are based on the same principle of hash generation, the Merkle-Damgård-Construction: First the message is padded to create an input of a certain fixed number of bits. Then this padded message is split into blocks of a fixed size and finally a compression function is applied to each block, using the output of the previous function and the next input block as input [ME79]. Both algorithms work with a round-based compression function with a fixed number of rounds.

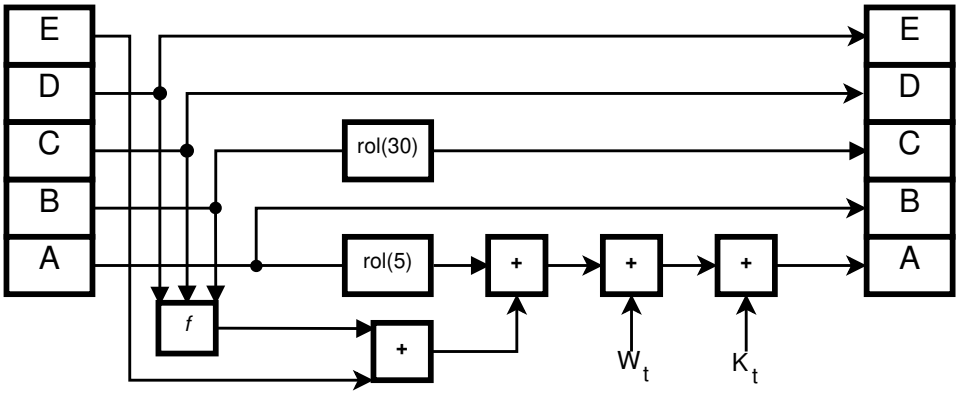


Fig. 1: SHA-1 round

SHA-1 works as follows (Fig. 1): The five 32 bit registers A–E get updated each round: Register A–D are copied to B–E with B being rotated by 30 bit. The round dependent `f`-function based on rotated values of the registers' old values, an earlier prepared array depending solely on the message (W_t) and a round-dependent constant (K_t) make up the new value of A. The concatenation of all registers form the hash after 80 rounds. SHA-1 utilizes the operations `AND`, `XOR`, `OR`, `ROL` (rotate left) and `ADD`. The algorithm uses the big-endian format.

In 2015 and 2017 two major attacks on SHA-1 got released (titled `SHAppening` and `SHAttered`), proving that it is possible to find collisions in a reasonable time when spending enough resources [St17][SKP15]. From that point on, SHA-1 was considered outdated and NIST recommended to stop using it in favor of SHA-2. However, due to the similar nature in construction it was feared that finding a fundamental weakness in the Merkle-Damgård-Construction would render SHA-2 unusable as well. This lead to the NIST hash function competition from 2007 to 2012 to find a suitable replacement for SHA-2.

2.2 SHA-3

In 2012 the competition ended with NIST announcing Keccak as SHA-3 [Dw15]. Developed by Bertoni et al., Keccak uses the sponge construction – in contrast to earlier SHA-algorithms. Keccak is also round-based. The sponge principle works by "absorbing" the message block by block with 1152 bit called rate and the remaining 448 bit serving as capacity. The rate part is determining the speed of hashing and the capacity part is serving as security parameter. The combination of both is called a Keccak-state, consisting out of 1600 bit for the most complex design of Keccak. The state is a 3D-array of 64-bit lanes in a 5x5x64 cuboid [Be] represented using the little-endian format. Each round, the state is run through the Keccak-f called function. Each Keccak-f function consists of the five steps called Theta, Rho, Pi, Chi

and Iota. These steps "mix up" the state by rotating, parity calculations, non-linear pattern calculations and symmetry-breaking actions (Fig. 2). After the whole message has been absorbed, a variable-length hash can be "squeezed" out (Fig. 3). Keccak uses basic and fast operations such as `OR`, `AND` and `XOR`.

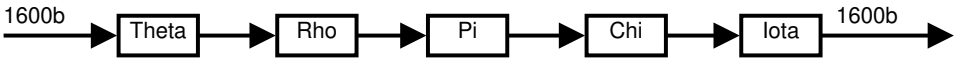


Fig. 2: SHA-3 round

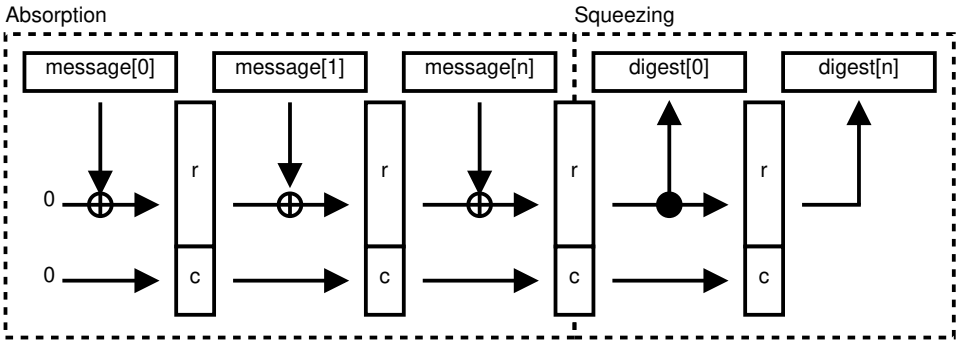


Fig. 3: SHA-3 architecture

Along with Keccak, other up-to-date hashing algorithms get used nowadays: `bcrypt` is one of the oldest examples still in use, using the expensive key setup from the cryptographic encryption algorithm `blowfish` [PM99]. Others include the already mentioned SHA-2-family or `scrypt` which is sequential memory-hard [PJ15] – exploiting the lack of resources in special hardware – and many other current algorithms along with the participants in the SHA-3 competition.

So far Keccak as SHA-3 has been shown to be resilient against collisions. However with collision finding techniques and a significant reduction of the number of rounds it is possible to compromise hashes [DDS12].

3 Experimental Setup

The most common computer architectures are GPUs, FPGAs, ASIC and General Purpose Processors (GPPs). Each of those architectures has different advantages. We wanted to have sufficient performance while being energy efficient. For our comparison, we chose to implement SHA-1 and SHA-3 on a GPP and an FPGA.

3.1 FPGA Design

To be able to brute-force strings, e. g. a password for a given hash, the design is separated into the following components (Fig. 4).

- The top component, called engine-master, responsible for dividing the workload and passing the found password to the CPU over the AXI interface
- The search engines, each responsible for a subpart of the search space
- The actual implementation of the algorithm, which can be exchanged for different algorithms
- A padder, which is dependent on the algorithm. It provides an incrementing password each clock cycle and pads according to the specification of the chosen algorithm.

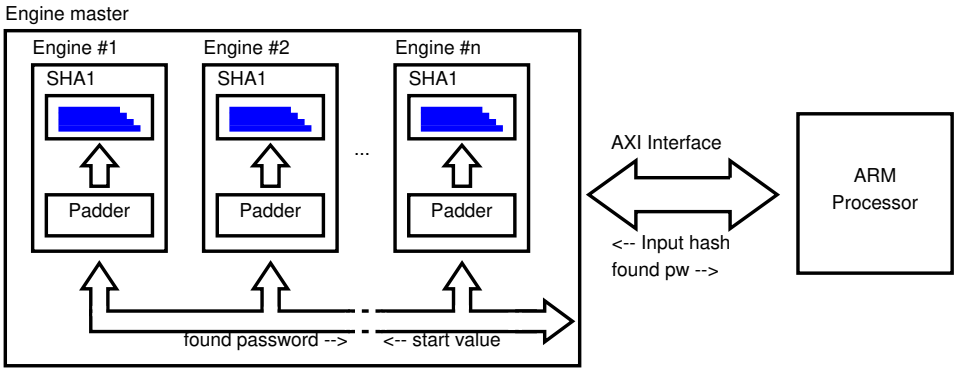


Fig. 4: Enginemaster with parallel engines

Depending on the available space and the desired performance, the number of search engines can be increased to crack more passwords in parallel.

3.1.1 Engine Design

The engines can optimized for either low area usage or high throughput. Trying to use the least resources possible per engine guarantees that many engines can be placed on the FPGA, each engine calculating one round of SHA-1 per clock cycle. Filling the board with these search engines while still maintaining a high clock frequency would split the search space in many small subparts which can quickly be searched.

However we noticed that because of the overhead of each engine resulting in high resource utilization, performance was very low and synchronization of the padder and engines was difficult and costly in terms of hardware resources. Also a large number of registers was needed for that approach. This lead us to abandon that approach and focus on pipelining a single engine. Such an approach leads to a high usage of resources such as LUTs but also results in very high throughput, generating at least one full hash per clock cycle.

Using multiple pipelined engines results in multiple hashes per clock cycle after the pipeline initialization. Each of those search engines contains its own padder, responsible for padding

a given password, according to the algorithm. The engine master divides the search space among the engines. We chose a simple approach of using one engine for all passwords of the length 1 to $n-1$ and the remaining engines for the password length of n characters. For example, with a maximum password length of five and an engine count of four the division is described in Tab. 1.

Engine	start value		end value	
0	␣	(ASCII 32)	~~~~	(ASCII 126)
1	␣␣␣␣␣	(ASCII 32)	>????	(ASCII 62, 63)
2	?????	(ASCII 63)	]^^^^	(ASCII 93, 94)
3	^^^^^	(ASCII 94)	~~~~~	(ASCII 126)

Tab. 1: Example of workload division to engines

3.1.2 SHA-1 Engine

In the SHA-1 engine the message gets padded to a multiple of 64 B first, with the length of the original message in bit in the last 1 B in our implementation. Generally, SHA-1 is able to hash blocks larger than 512 bit. However, due to an average password needing less than one block, we limit our design to process only one block. Each engine contains a pipeline with 80 stages, each calculating one round of SHA-1.

3.2 SHA-3 Engine

Our goal was to design the architecture to be as modular as possible, to simplify swapping algorithms. However SHA-3 uses a different padder, which adds the value `0x06` after the message and the `0x80` at the end (depending on the message size these two values might be merged into one byte `0x86`). SHA3-256 implements the most complex design of the original Keccak algorithm, which uses a block size of 1600 bit, and the earlier explained padding scheme. We pipelined the SHA-3-algorithm published by the Keccak team[Be11] to increase performance and alternated it to fit into our architecture. However the sheer number of status bits in SHA-3 make the whole algorithm very resource-intensive leading to a high resource utilization for each engine.

3.3 Software implementation

Using the C-language and Intel intrinsics to target the `Advanced Vector Extensions2 (avx2)` and multithreading, we wanted to achieve the highest possible performance. The code was compiled with `gcc 7.3.0 -O3 -mavx2`. Every 256 bit vector was split into eight

32 bit (SHA-1) or four 64 bit (SHA-3) lanes. Each lane is responsible for one hash, meaning that eight SHA-1 or four SHA-3 hashes are computed simultaneously. This concept is similar to the FPGA multiparallel engine approach with the threads being the engines. As our experimental setup, we had a dual-CPU server driven by two `Intel® Xeon® E5-2680 v3 Processors` with a clock frequency of 2.5 GHz. Each CPU has 12 cores. With 24 cores and hyperthreading, 48 threads are executed at the same time.

4 Experimental Results

We evaluated both design in terms of performance, hardware cost and energy efficiency. Additionally, a comparison was done with other state-of-the-art hardware.

4.1 Software solution

Tab. 2 shows that SHA-1 is approximately 28x faster than SHA-3. With a factor of 2x coming from twice as many hashes in each vector register. The remaining factor of 14 stems from the high number of status bits required for SHA-3 besides the higher complexity of the algorithm itself. SHA-3 works on $64 \cdot 25 = 1600$ status bits for each processing step. This results in 25 vectors required for storing the status bits only. With AVX2 supporting up to 16 vectors, data between cache and vector registers has to be swapped, which is time consuming. Consequently, if we increase the number of threads, we also increase the hash rate. Increasing the thread count works well until we hit a peak at 48 threads, hashing at a rate of 1057.5 Mhashes/s (SHA-1) and 37.1 Mhashes/s (SHA-3), respectively.

	SHA-1			SHA-3		
# Threads	Hashrate [Mhashes/s]	Power [W]	Efficiency [Mhashes/J]	Hashrate [Mhashes/s]	Power [W]	Efficiency [Mhashes/J]
1	21.8	177	0.12	20.4	180	0.11
2	43.7	199	0.21	20.5	201	0.10
24	643.2	299	2.15	28.8	329	0.08
48	1057.5	313	3.38	37.1	339	0.11

Tab. 2: Software Implementation Results SHA-1/SHA-3

4.2 FPGA-based solution

We used a `Xilinx Ultrascale+ MPSoC Device with Quad-Core Arm Cortex-A53` FPGA. The `2017.3 Vivado` version was used to synthesize and implement our VHDL design.

4.2.1 Performance

We implemented the design with varying engine numbers and tried several clock speeds to generate the highest possible hash rate. Tab. 3 depicts these results for each engine with its corresponding frequency. In our multiparallel pipelined architecture, one hash is generated per clock cycle. This translates directly to the hash rate being a multiplication of the number of engines multiplied by the clock frequency. Generally, the more hardware resources are used, the smaller the clock frequency will be. Increasing the number of SHA-1 engines first gave us a higher hash rate until we reached the sweet spot at 12 engines where the clock frequency was at 370.4 MHz, giving us a total hash rate of 4444.8 Mhashes/s.

Due to the search space division, the SHA-3 hash rate was still rising with four engines but five engines could not be implemented on the board. So the best performance for SHA-3 was accomplished with four engines and a corresponding clock frequency of 204.0 MHz, contributing to a hash rate of 816.0 Mhashes/s. As shown in Tab. 2 and Tab. 3, SHA-1 not

	# Eng.	Freq. [MHz]	LUTs [usage]		Flip-Flops [usage]		Hashrate [Mhashes/s]	Power[1] [W]	Efficiency [Mhashes/J]
SHA-1	1	543.5	14.6 k	5.3 %	26.5 k	4.8 %	543.5	4.2	129.4
	8	384.6	114.0 k	41.6 %	210.0 k	38.3 %	3076.8	19.5	157.8
	9	384.6	128.9 k	47.0 %	236.2 k	43.1 %	3461.4	21.5	161.0
	12	370.4	170.1 k	62.0 %	314.9 k	57.4 %	4444.8	27.3	162.8
	16	259.7	224.9 k	82.1 %	419.7 k	76.6 %	4155.2	28.6	145.3
	17	256.4	238.9 k	87.2 %	445.9 k	81.4 %	4358.8	31.2	139.7
SHA-3	1	277.7	49.0 k	17.9 %	36.3 k	6.6 %	277.7	12.6	22.0
	2	222.2	97.0 k	35.4 %	72.4 k	13.2 %	444.4	18.7	23.8
	3	208.3	145.5 k	53.1 %	108.5 k	19.8 %	624.9	25.6	24.4
	4	204.0	193.9 k	70.8 %	144.6 k	26.4 %	816.0	34.2	23.9

Tab. 3: Hardware Implementation Results SHA-1/SHA-3

only has a higher hash rate than SHA-3 in software, but also in the hardware implementation, the difference being an additionally speedup of 5.5×. This observation is explained by the higher complexity of the SHA-3 algorithm. SHA-1 is a straight-forward algorithm to implement with only 160 status bits. SHA-3 on the other hand has ten times that many status bits while also having more complex computations during every round.

[1] According to Vivado Implementation report

4.2.2 Hardware Cost

Having a look at the hardware costs for our implementations, we see that the Look-Up-Table and Flip-flops usage increases linearly with the number of engines. This means the more LUTs and FFs we have, the more engines we can implement on the FPGA. SHA-3 has three times the LUT usage as SHA-1 with 49k compared to 14.6k, respectively. Higher hardware usage lets the frequency drop while also decreasing the number of search engines. Hence decreasing the overall hash rate. At the fastest hash rate, SHA-1 uses 62.0 % of all LUTs and 57.4 % of all FFs. Meaning that more engines could have been implemented, but the overall hash rate was dropping due to the frequency loss. Also SHA-1 balances the usage of FFs and LUTs, ensuring to not have a bottleneck with either. SHA-3 uses a lot more LUTs (70.8 %) than FFs (26.4 %). This behavior is also linked to the more complex round design of SHA-3.

4.2.3 Energy Efficiency

We measured the energy efficiency by dividing the maximum hash rate by the power consumption. As shown in Tab. 3, the FPGA uses 27.3 W (SHA-1) and 34.2 W (SHA-3) at the highest hash rate. This results in hashes per joule of 162.8 Mhashes/J (SHA-1) and 23.9 Mhashes/J (SHA-3). This illustrates that SHA-3 is 6.8× less energy efficient. Comparing the energy efficiency of both algorithms on the CPU fortifies the suspicion of SHA-1 being the more energy efficient algorithm (Tab. 2). SHA-1 and SHA-3 process 3.38 Mhashes/J and 0.11 Mhashes/J respectively, which corresponds to a 30.7× difference in energy efficiency.

4.3 Comparison with state-of-the-art Hardware

A GTX 1080 Ti calculates SHA-1 hashes at an approximate rate of 11.6 Ghashes/s [Ic17]. Our maximum hash rate of 4.448 Ghashes/s is about 2.5 times slower. Adding the power usage of a GTX 1080 Ti with 250 W and the power usage for the FPGA of 27.3 W, this means that our FPGA solution is 3× more energy efficient. Dat et. al. achieved a throughput of around 20 GB/s for SHA-3 ([DIK17]). They used a GTX 1080 and CUDA to obtain their numbers. Our design has a maximum data throughput of about 40.8 GB/s. This means that our designs on an FPGA are about twice as fast. Comparing our software and FPGA solution shows that the FPGA has a 4.2× better performance than the GPP for SHA-1. With SHA-3 the FPGA is even 21.9× faster compared with the software implementation.

4.4 Password Cracking Time

Hashing is also used to store passwords. Tab. 4 depicts the longest possible times of our designs needed to crack a password of a certain length. The two most important criteria

	SHA-1		SHA-3	
PW length	FPGA	CPU	FPGA	CPU
≤ 3	<0.1 s	<0.1 s	<0.1 s	<0.1 s
4	<0.1 s	<0.1 s	0.1 s	2.10 s
5	1.65 s	6.94 s	8.99 s	3.29 min
6	2.59 min	10.87 min	14.09 min	5.15 h
7	4.05 h	17.03 h	22.08 h	20.18 d
8	15.87 d	66.72 d	86.46 d	1896.56 d

Tab. 4: Time to crack password with SHA-1/SHA-3 (Worst Case)

of password security are the password length and the value range of password characters. While passwords with six or less characters can be cracked in minutes, password of eight or more characters can take days to crack. Reducing the amount of characters from 94 to 62, by ignoring all special characters, results in a 27.9× loss of cracking time.

Furthermore, finding a password on a CPU takes a lot more time than on an FPGA. SHA-3 has an even higher performance loss comparing hardware with software than SHA-1.

	SHA-1	SHA-3	Speedup
Software	1057.5 Mhashes/s	37.2 Mhashes/s	28.4×
Hardware	4444.8 Mhashes/s	816.0 Mhashes/s	5.5×

Tab. 5: Results SHA-1/SHA-3

5 Conclusions

We evaluated the resilience of SHA-1 and SHA-3 hashing algorithms against brute-force attacks. The performance on a `Zynq Ultrascale+ MPSoC` is 4.2× higher than with a 24-core processor (*Xeon E5-2680 v3*) for SHA-1 and 21.9× higher for SHA-3 due to the modified construction algorithm. Higher algorithmic complexity brute-force attacks result in 5.5× longer runtime with SHA-3 than with SHA-1. Additionally, while passwords with a length of six characters can be found in minutes, days are required for passwords with at least eight characters. This underlines the need for passwords of sufficient length and the usage of recent hash algorithms.

References

[Be] Bertoni, G.; Daemen, J.; Hoffert, S.; Peeters, M.; Assche, G. V.; Keer, R. V.: Keccak specifications summary, URL: https://keccak.team/keccak_specs_summary.html.

[Be11] Bertoni, G.; Daemen, J.; Hoffert, S.; Peeters, M.; Assche, G. V.; Keer, R. V.: Hardware implementation in VHDL, Jan. 31, 2011, URL: https://keccak.team/obsolete/KeccakVHDL-3.0.zip.

[DDS12] Dinur, I.; Dunkelman, O.; Shamir, A.: New Attacks on Keccak-224 and Keccak-256. In: Fast Software Encryption. Springer Berlin Heidelberg, pp. 442–461, 2012.

[DIK17] Dat, T. N.; Iwai, K.; Kurokawa, T.: Implementation of High Speed Hash Function Keccak Using CUDA on GTX 1080. In: 2017 Fifth International Symposium on Computing and Networking (CANDAR). IEEE, Nov. 2017.

[Dw15] Dworkin, M. J.: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, tech. rep., July 2015.

[Ic17] Ickler, K. R.: Hashcat Benchmarks for Nvidia GTX 1080TI & GTX 1070 Hashcat Benchmarks, June 20, 2017, URL: https://www.blackhillsinfosec.com/hashcat-benchmarks-nvidia-gtx-1080ti-gtx-1070-hashcat-benchmarks/.

[ME79] MERKLE, R.: Secrecy, Authentication, and Public Key Systems. Ph. D. Thesis, Stanford University/, 1979, URL: https://ci.nii.ac.jp/naid/10020925339/en/.

[na95] national: FIPS PUB 180-1: Secure Hash Standard. Gaithersburg, MD, USA, 1995.

[PJ15] Percival, C.; Josefsson, S.: The Scrypt Password-Based Key Derivation Function (2012). URL http://tools. ietf. org/html/josefsson-scrypt-kdf-00. txt 2/10, 2015.

[PM99] Provos, N.; Mazieres, D.: Bcrypt algorithm. In. USENIX, 1999.

[Ri92] Rivest, R.: The MD5 Message-Digest Algorithm, tech. rep., Apr. 1992.

[SKP15] Stevens, M.; Karpman, P.; Peyrin, T.: Freestart collision for full SHA-1, Cryptology ePrint Archive, Report 2015/967, https://eprint.iacr.org/2015/967, 2015.

[St17] Stevens, M.; Bursztein, E.; Karpman, P.; Albertini, A.; Markov, Y.: The first collision for full SHA-1. In: Annual International Cryptology Conference. Springer, pp. 570–596, 2017.

B. Juurlink, W. Karl (Hrsg.)orkshop,
Proceedings 28th PARS Workshop

Evaluating the Usability of Asynchronous Runge-Kutta Methods for Solving ODEs

Christopher Greene[1], Markus Hoffmann[1]

Abstract: Combining asynchronous methods with scientific computing is a great challenge. In this paper we make the attempt to combine such methods with a ODE solver. Although the results are not on point for giving us a fully usable asynchronous method, this paper shows the direction of the needed development to get such an asynchronous method.

Keywords: Differential Equations; Asynchronous Computations; Runge-Kutta Methods

1 Introduction

Scientific computing poses a difficult challenge for people from different domains, especially in order to find a suitable trade-off between desired solution quality and computational effort. Even the high parallel capabilities of todays hardware do not lead to a significant reduction of these challenges because of the increasing dimensions of current problems and low parallelization potentials of the algorithms, especially when it comes to solving ordinary differential equations (ODEs). Additionally, high accuracy is often inevitable for scientific computing.

Hence, at first glance, it seems counterproductive to marry approximation strategies like computations with expected data races with scientific computing. However, there is already some successful work that introduces such strategies into scientific computing [ADQ15],[Zh14]. Asynchronous parallelization methods in particular, which can be compared with relaxed synchronization, are well-known in numerics and show a high efficiency on GPUs [ACD15].Therefore, this paper evaluates the application of these methods on solvers for ODEs and widenings the understanding of the potential and limits of these methods. It also gives an insight on the requirements a numerical method has to meet to be a good candidate for applying asynchronous parallelization methods.

[1] Karlsruhe Institute of Technology, Chair of Computer Architecture and Parallel Processing, Kaiserstr. 12 / 76131 Karlsruhe, Germany markus.hoffmann@kit.edu

1.1 Current Status

Small et al. [Sm13] describe a method for solving sparsely linked ODEs, given they can be arranged in a directed tree structure. The nodes of this tree are distributed on a system with distributed memory for parallel calculation, solving the tree structure starting with the outmost leaves and spreading the solutions to connected nodes, which then calculate their solutions.

Barros [Ba15] work implements an individual step size control for all equations on an asynchronous implementation, utilizing error estimation. Threads are only notified of change, if an error exceeds a given maximum.

Another approach in solving ODEs is the QSS-method by Kofman [KJ01], which divides the value-axis into quanta instead of the time-axis. The slope of an equation is calculated and then assumed as constant, until a quantum is reached, where the slope is recalculated. This method arose from event-driven architectures, where state changes are more important than the exact value to any given time.

1.2 Methodology of the Evaluation

We analyze the usage of asynchronous methods by targeting different systems of ODEs with a widely used method, the classical Runge–Kutta method. Firstly, we take a look into the used solving methods and systems (see Section 2). Then, we introduce the used strategies for asynchronous computing and the basic ideas behind the implementation in Section 3. To note, we analyze the applicability of asynchronous approaches, but we do not provide a run-time approach that controls the quality. With Section 4 we compare our different approaches regarding their execution times and the relative error. This evaluation aims to answer the following questions: Is it possible to create a fast asynchronous ODE solver that keeps accuracy on an acceptable level? How can such a solver be improved to increase the performance of these methods?

1.3 Main Findings

Based on the outcome of our experiments, the following conclusions can be drawn:

- **Conclusion 1:** Yes, it is possible to create an asynchronous solver with acceptable accuracy and speed-up.
- **Conclusion 2:** The methods shown in this paper will not be used in practice, because they can be substituted by equivalent and well-known synchronous solvers.
- **Conclusion 3:** There might be a way to create more useful asynchronous methods, and we have a good idea of the direction of development.

2 Mathematical Background

2.1 Soving Systems of Coupled ODEs

A nearly universally applicable and widely used class of methods for solving ordinary differential equations (ODEs), as shown in equation (1), is the class of Runge-Kutta methods. The simplest method of this class is the Euler method [Ha01], referenced as RK1 in this paper. For a given ODE

$$\dot{u}(t) = f(t, u(t)) \tag{1}$$

and a problem dependent chosen starting value u_0, RK1 is specified as

$$u(t_{n+1}) = u(t_n) + hf(t_n, u(t_n)), \qquad u(t_0) = u_0, \tag{2}$$

where h is a discretization parameter for the time axis, which means $t_{n+1} = t_n + h$ for equidistant discretization. Based on this specification, the class of Runge-Kutta methods can be completed by adding additional supporting points within (t_n, t_{n+1}) and combining the evaluation of (1) on these points with the help of weighting factors. One of the most famous representatives of the class constructed in this way is the RK4 method (also known as the method of Runge and Kutta) [Ha01] as given in (3):

$$u(t_{n+1}) = u(t_n) + h\left(\frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4\right), \qquad u(t_0) = u_0, \tag{3}$$

$$k_1 = f(t_n, u),\ k_2 = f\left(t_n + \frac{h}{2}, u + \frac{h}{2}k_1\right),\ k_3 = f\left(t_n + \frac{h}{2}, u + \frac{h}{2}k_2\right),\ k_4 = f(t_n + h, u + hk_3)$$

From the mathematical point of view, the usage RK4 goes along with lower error rates (which basically allows greater time steps) compared to RK1, whereas the computer scientists perspective shows the need of more computing effort within RK4, leading to a higher computation time per step. However, the most important aspect for us is the parallelization potential.

We address this aspect by solving a system of ODEs instead of a single one. Moreover, to show the influence of synchronization issues, we require a system of coupled ODEs [Ha01] as shown in (4):

$$\begin{aligned} \dot{u}_1(t) &= f_1(t, u_1(t), u_2(t), \ldots, u_{n-1}(t), u_n(t)) \\ &\vdots \\ \dot{u}_n(t) &= f_n(t, u_1(t), u_2(t), \ldots, u_{n-1}(t), u_n(t)). \end{aligned} \tag{4}$$

The most intuitive way, and also the way with the lowest amount of needed system characteristics, for parallelization of the Runge-Kutta methods is to distribute the equations of (4) to the available threads. This results in synchronization requirements after each

step for RK1 and additionally after each k_i for RK4, which means a factor of 5 for the synchronization effort compared to RK1. In conclusion, without any specific knowledge about the system of coupled ODEs, RK4 allows greater time steps on the cost of computing time for a single step in each equation and synchronization effort per step over the whole system compared to an equivalent RK1.

2.2 Coupled ODEs used for testing

For testing the potential of asynchronous Runge-Kutta methods we use two systems of ODEs. The first one is the Lotka-Volterra model (predator-prey equations) [Ol94], given as

$$\dot{u}_i = u_i \left(b_i + \sum_{j=1}^{m} a_{ij} u_j \right), \qquad i = 1, 2, \ldots, m, \tag{5}$$

where m represents the number of species, b_i defines the birth rate of species i, and a_{ij} addresses the way of interaction between species i and species j.

For more advanced testing we used a single track model [Ge05] for simulating the behavior of a car while moving. This model is given by the following equations:

$$\begin{aligned}
\dot{x}(t) &= v(t)\cos(\psi(t) - \alpha(t)), \qquad \dot{y}(t) = v(t)\sin(\psi(t) - \alpha(t)) \\
\dot{v}(t) &= \frac{1}{m}[(F_{lr} - F_{Ax})\cos(\alpha(t)) + F_{lf}\cos(\delta(t) + \alpha(t)) \\
&\qquad - (F_{sr} - F_{Ay})\sin(\alpha(t)) - F_{sf}\sin(\delta(t) + \alpha(t))] \\
\dot{\psi}(t) &= w_z(t), \qquad \dot{\delta}(t) = i_s \dot{w}_\delta(t) \\
\dot{\alpha}(t) &= w_z(t) - \frac{1}{mv(t)}[(F_{lr} - F_{Ax})\sin(\alpha(t)) + F_{lf}\sin(\delta(t) + \alpha(t)) \\
&\qquad + (F_{sr} - F_{Ay})\cos(\alpha(t)) + F_{sf}\cos(\delta(t) + \alpha(t))] \\
\dot{w}_z(t) &= \frac{1}{I_{zz}}[F_{sf} l_f \cos(\delta(t)) + F_{lf} l_f \sin(\delta(t)) - F_{sr} l_r - F_{Ay} e_{sp}]\,,
\end{aligned} \tag{6}$$

where x and y are representing the global position of the car, v gives the moving speed, ψ / w_z and α are showing the (global and local) angles of the moving direction, and δ the the general direction of the (steerable) front wheels. This model can be parametrized by $w_\delta(t)$ (position of the steering wheel), $F_B(t)$ (used braking force), $\phi(t)$ (acceleration), and $\mu(t)$ (used gear).

We used the first (easy scalable) model to show, that basic assumptions about the numerical characteristics and estimations from a computer scientists perspective are correct for our methods. Because of its complexity, level of coupling, and computational effort the second model can be used to test the practical relevance of our methods.

3 Strategies for Asynchronous ODE Solvers

Avoiding synchronization within any numerical method comes down to two basic approaches. The most intuitive option is to avoid every single synchronization. But this may result in uncontrollable errors or convergence problems which leads to the idea of selective synchronization cancelling. For RK4 we decided to evaluate both approaches to find the most suitable implementation.

3.1 Semi-Asynchronous Runge-Kutta Method

As seen before, a synchronized Runge-Kutta implementation needs to calculate 4 k_i for each of n differential equations. These k_i are showing dependencies to the previously computed k_i of the same step and, with reference to the coupling between the equations, synchronization is needed to provide all needed data for each k_i. Additionally, synchronization is needed at the end of each step to compute the next solution. The basic idea behind the semi-asynchronous RK4 implementation is the cancelling of synchronization while computing the k_i, but not at the end of an iteration step. We implemented this by setting up an array for all needed k_i of a single step (resulting in an array size of $n * 4$) and reusing this array for each step without resetting. This means faster (with repect to computation time) equations will use old k_i-data from slower equations. However, with the help of the synchronization at the end of each step, a faster equation will get these k_i-data from the previous step that hopefully do not differ too much from the k_i of the correct step. Together with a suitable chosen discretization parameter h this hopefully will result in small approximation errors. On the other hand, $\frac{4}{5}$ of synchronization is avoided.

3.2 Full Asynchronous Runge-Kutta Method

The idea of the Full Asynchronous Runge-Kutta Method is to remove even the last synchronizing structure. The main issue here is to avoid exploding approximation errors. For that, we implemented a scheduler that holds all equations in a queue. This scheduler is equipped with distance parameter d to restrict the iteration step difference of the slowest equation (t_{min}) and the fastest equation (t_{max}) by the condition $t_{max} - t_{min} \leq d$. Additionally, we need to set up a policy to find suitable k_i-data. For the asynchronous implementation it is done by saving all k_i data from all equations and all steps. Thus it is possible to find the most suitable k_i-data for each equation: the newest available one for the fast equations and for the slow equations we can use the computed k_i from the actual step of the slow equation. Obviously, the approximation error rises with badly chosen k_i-data. That is why we want to mention, that we tried to make sure to give an equation the "closest-to-its-own-step" k_i-data from each other equation resulting in an implementation with a registry system for all computed k_i. However, missing synchronization also reduces the efficiency of such a system but it was the most efficient implementation we found with respect to the approximation error.

4 Experiments

We run all the experiments on two Intel Xeon CPU E5-2650 v4 processors (2.2GHz, 24 cores each) providing 64 GB of main memory. A synchronous and parallel version of the RK4 solver executed using n threads for n equations is our base line. The basic discretization parameter h is set to 0.05 for all RK4 and RK1 measurements, if not mentioned otherwise. We set the iteration count to 30000 which provides enough iterations to take a look into the long therm effects of our approaches.

4.1 Evaluation Metrics

For accuracy, we thought of calculating the standard relative error. Unfortunately, a not negligible amount of the exact solutions are very close to 0. To avoid dividing by 0, we changed the error calculations to a normalized L^1-based measurement, given as

$$\hat{E}_{rel} = 2 * \frac{x_{exact} - x_{approx}}{|x_{exact}| + |x_{approx}|}, \qquad E_{rel} = \frac{1}{2}|\hat{E}_{rel}|.$$

In order to improve the visualization, we decided to plot the average error of all equations at a certain time:

$$E = \sum_{i=1}^{n} \frac{E_{rel_i}}{n}, \qquad \text{for } n \text{ equations.}$$

At this point it is important to mention that this error measurement comes with a great problem. A small absolute error that occurs close to zero might have a much bigger relative error than an error of the same size that occurs not in a near range around zero. This leads to peaks in the relative error and pointing out the maximum relative error would lead to biased results. Therefore, we decided to show the average error in every figure, because these peaks do not have a great impact on the average error and although the maximum error would be more interesting, our conclusions can also derived from the average error.

4.2 Analysis of (Semi-)Asynchronous Runge-Kutta Method

First of all we want to show the basic behaviour of our methods. For the Lotka-Volterra model (LVM) we evaluated systems of 2 species up to systems of 10 species. We set the parameters in a way that each system can be classified as one out of two classes. The first class are asymptotic systems in the meaning that all equations are running against a single value with only some oscillations at the beginning. The second class are periodic oscillating systems with different frequencies and amplitudes. Figure 1 shows the error of a representitve[2] of each class: LVM10 for the oscillating systems, LVM9 for the asymptotic ones. Additionally, the error for the single track model (STM) is visualized. This specific

[2] We are focusing on these systems because of equivalent results for all other class members.

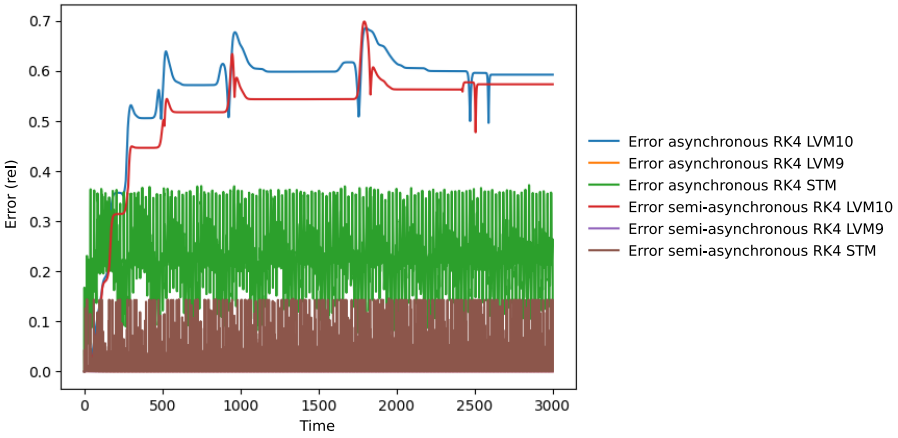


Abb. 1: Accuracy of representitives of system classes over time.

model describes a car running in a circle with some acceleration at the beginning. Such a parameter set leads to a mixture of different equations within the system: x and y are oscillating, v has an asymptotic behaviour, ψ increases constantly, and so on. For the full asynchronous method the distance parameter d is initially set to 1 for all models.

Obviously, the error for the asymptotic LVM9 vanishes immediately, while the error for the oscillating LVM10 shows increasing errors over the time. The error for the STM oscillates very fast within a rang between LV9 and LV10. Additionally, the semi-asynchronous method shows smaller error rates compared to the full asynchronous method for almost all models and points in time.

We do not have a special picture of the run times here because they are very unsurprising: for the LVM the speed-up is around 1.5 for the semi-asynchronous method and around 1.3 for the asynchronous method. For the STM the semi-asynchronous speed-up is 1.3 and the asynchronous speed-up 1.1. The most interesting thing about the run times is the unexpected slow asynchronous method. The reason for this is the combination of using enough threads for all equations and a scheduler based implementation which limits the runtime by the slowest equation and puts additional administrative effort on top. Therefore, we added some experiments with variable thread numbers, variable distance parameter, and variating discretization parameter, especially for the STM. Figure 2 shows the results of the last mentioned. The most important part of this figure is the POI-line that shows the fixed discretization of all synchronous methods. We leave out the figures for the other experiments as they are exactly like expected: For a falling number of computing threads, all methods are loosing speed. Because of the low number of needed threads for full parallelization,

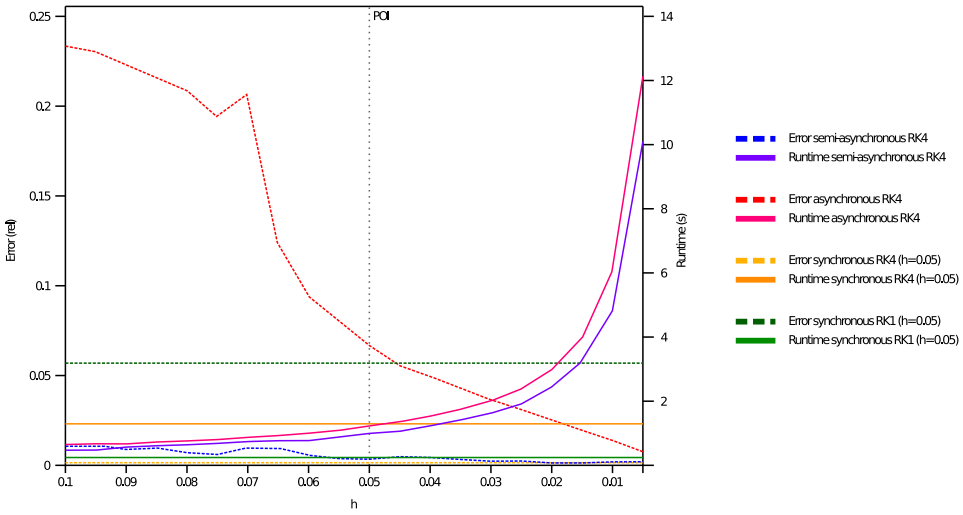


Abb. 2: STM for variating discretizations. The POI (point of interest) marks the fixed discretization for the synchronous methods.

the runtime increases for all methods to the same extent, which made this experiment not evaluable. Changing the distance parameter gives a slow speed up for all models while increasing the error enormously, especially (but not only) in case of the STM. Therefore, for discussing these results we are focusing on the error over time measurements and the method comparison for the STM.

4.3 Discussion

For result discussion the LVM gives us the most basic insight view. Looking at figure 1 a basic expectation can be confirmed: Models with high frequency oscillating equations are more error-prone to asynchronous methods than models with low frequency equations. Another expectation can be confirmed by looking at LVM with different numbers of equations. A model with more equations is more robust to errors induced by asynchronous methods. To give the most impressive example: The error for the asynchronous LVM2 can't be computed because the result values are exploding to infinity, while an asynchronous LVM10 creates a great but not exploding error. This behaviour of increasing error resistance for increasing number of equations can be seen for all asynchronous methods and oscillating systems.

For further discussion we now have to change the model to the STM because the LVM is too simple to compute to get robust results. Looking at the POI-line of figure 2, we can see, that both asynchronous methods are faster than both asynchronous methods. Focusing on the semi-asynchronous method we can also see very suitable error that is only slightly

higher than the error of the equivalent RK4. This leads us to our first conclusion: Yes, it is possible to create a usable asynchronous method. Unfortunately, the speed-up at this point is very low. That is why we think one can use RK4 with a specific $h > 0.05$ to get the same results. This gives us the second conclusion: This method seems not to be useful in practice. We made this figure also with the intention to see if a slightly decreased h for the semi-asynchronous method can help to get better error rates by providing a speed-up to the synchronous method for $h = 0.05$. Unfortunately, the runtime rises much faster than the error rate drops. And again, the to low initial speed-up shows that there is no potential for improvement in this direction for this model.

Looking at the asynchronous method for this model gives an impression how difficult it is to get the error under control. Although we have chosen the lowest d, which results in nearly (but not exactly) equivalent method to the semi-asynchronous method, the error is even higher than the error of the equivalent RK1 while the runtime is only slightly better than the runtime of RK4. As for the semi-asynchronous method, we hoped for a higher speed-up which would allow a decreasing h to stabilize the error, especially because the error is decreasing much faster than the error for the semi-asynchronous method.

To put it in a nut shell, we can see that basic expectations from the numerical point of view and from a computer scientists point of view are met by the asynchronous methods, which helps to predict the behaviour of such methods. Additionally, we could see, that it is possible to create an asynchronous method that gives speed-up on the cost of a controllable increase of the approximation error. Unfortunately, this speed-up has to be increased a lot to get a useful method for practical application.

5 Conclusion and Future Directions

In this paper we evaluated the possibilities of applying asynchronous methods to an ODE solver. For that, we applied a semi-asynchronous and a full asynchronous method to a Runge-Kutta solver and compared the run times and error rates with the equivalent synchronous method as well as the Euler method. We showed that a speed-up is gainable for both asynchronous methods, but we could also see that this speed-up needs to be much higher to get practically useful methods. Also, decreasing the resulting approximation error is a good option for getting useful methods, especially in case of the full asynchronous method.

We also get an impression of the numerical characteristics of these methods, which leads us to two ideas for future development. The first idea is based on the speed-up. In a future work we should test our actual methods on systems with a huge amount of equations. Based on the implementation of our methods, they will increase their speed-up drastically when the equation-prozessor-ratio is getting worse. This speed-up would give us the possibility to adjust the discretization for the asynchronous methods which hopefully leads to much better results.

The second idea is based on the error rate. Our implementation, especially the full asynchronous one, only uses old and therefore bad values if the actually needed values are not available. But if we take a view beyond the horizon there are, for example, the implicit methods using also future values to lower their error rates and become more stable. Maybe it is possible to use this idea for faster equations (because there might be enough time to compensate the additional computing effort) and also for slower equations (because we will get future values with no additional effort) of a system to lower the error rates. Also multi-step methods or even the QSS-methods might be something to think about in this context.

In conclusion, we do have to do a lot of more work to get really useful asynchronous methods for solving ODEs.

Literatur

[ACD15] Anzt, H.; Chow, E.; Dongarra, J.: Iterative Sparse Triangular Solves for Preconditioning. In (Träff, J. L.; Hunold, S.; Versaci, F., Hrsg.): Euro-Par 2015: Parallel Processing. Bd. 9233, Lecture Notes in Computer Science, Springer Berlin Heidelberg, S. 650–661, 2015, ISBN: 978-3-662-48095-3.

[ADQ15] Anzt, H.; Dongarra, J.; Quintana-Ortı, E. S.: Adaptive precision solvers for sparse linear systems. In: Proceedings of the 3rd International Workshop on Energy Efficient Supercomputing. ACM, S. 2, 2015.

[Ba15] Barros, F.: Asynchronous, Polynomial ODE Solvers Based on Error Estimation. In: Proceedings of the Symposium on Theory of Modeling & Simulation: DEVS Integrative M&S Symposium. DEVS '15, Society for Computer Simulation International, Alexandria, Virginia, S. 115–121, 2015.

[Ge05] Gerdts, M.: Solving mixed-integer optimal control problems by branch & bound: a case study from automobile test-driving with gear shift. Optimal Control Applications and Methods 26/1, S. 1–18, 2005.

[Ha01] Hazewinkel, M.: Encyclopedia of Mathematics. Springer Science+Business Media B.V. / Kluwer Academic Publishers, 2001.

[KJ01] Kofman, E.; Junco, S.: Quantized State Systems. A DEVS Approach for Continuous System Simulation. Transactions of SCS 18/3, S. 123–132, 2001.

[Ol94] Olek, S.: An Accurate Solution to the Multispecies Lotka–Volterra Equations. SIAM Review 36/3, S. 480–488, 1994.

[Sm13] Small, S. J.; Jay, L. O.; Mantilla, R.; Curtu, R.; Cunha, L. K.; Fonley, M.; Krajewski, W. F.: An asynchronous solver for systems of ODEs linked by a directed tree structure. Advances in Water Resources 53/, S. 23–32, 2013.

[Zh14] Zhang, Q.; Yuan, F.; Ye, R.; Xu, Q.: Approxit: An approximate computing framework for iterative methods. In: Proceedings of the 51st Annual Design Automation Conference. ACM, S. 1–6, 2014.

Reducing DRAM Accesses through Pseudo-Channel Mode

Farzaneh Salehiminapour, Jan Lucas, Matthias Goebel, Ben Juurlink[1]

Abstract:

Applications once exclusive to high-performance computing are now common in systems ranging from mobile devices to clusters. They typically require large amounts of memory bandwidth. The graphic DRAM interface standards GDDR5X and GDDR6 are new DRAM technologies that promise to almost doubled data rates compared to GDDR5. However, these higher data rates require a longer burst length of 16 words. This would typically increase the memory access granularity. However, GDDR5X and GDDR6 support a feature called pseudo-channel mode. In pseudo-channel mode, the memory is split into two 16-bit pseudo channels. This split keeps the memory access granularity constant compared to GDDR5. However, the pseudo channels are not fully independent channels. Two accesses can be performed at the same time but access type, bank, and page must match, while column address can be selected separately for each pseudo channel. With this restriction, we argue that GDDR5X can best be seen as a GDDR5 memory that allows performing an additional request to the same page without extra cost. Therefore, we propose a DRAM buffer scheduling algorithm to make effective use of the pseudo-channel mode and the additional memory bandwidth offered by GDDR5X. Compared to the GDDR5X regular mode, our proposed algorithm achieves 12.5% to 18% memory access reduction on average in pseudo-channel mode.

Keywords: Memory Management; Memory Bandwidth; GDDR5X; GDDR6

1 Introduction

Over the last decade, due to the increasing number of memory-intensive applications, DRAM bandwidth is known as one of the major performance bottlenecks in modern multi-core systems. On one hand, new generations of DRAM technologies have been designed to improve memory performance. For example, the Double Data Rate (DDR) standard was introduced to increase memory bandwidth by transferring two data words per clock cycle. Besides, to achieve further bandwidth, new I/O features have been added to modern DRAM standards such as Graphic Double Data Rate 5 (GDDR5) and its modified version GDDR5X. In comparison to GDDR5, GDDR5X employs a new I/O feature which can increase the maximum data rate per pin by up to 14Gb/s/pin [Br18] compared to 7Gb/s/pin [JE16a] for GDDR5. However, as the speed of the internal memory array is not increased the memory burst size is increased to 16 words. With this burst size, 64 bytes are transferred in each

[1] Technische Universität Berlin, Architektur eingebetteter Systeme(AES), Einsteinufer 17, 10587 Berlin, Germany
salehiminapour,j.lucas,m.goebel,b.juurlink@tu-berlin.de

read or write transaction, if using a regular 32-bit wide interface. To reduce the issue of this large memory access granularity, GDDR5X features a pseudo-channel mode which enables the memory controller to fetch two simultaneous memory access with 32-bytes access granularity from the two different pseudo-channels, provided these accesses have the same access type, rank, bank and page [JE16b]. Pseudo-Channel Mode is not limited to GDDR5X, GDDR6 [JE17] is also equipped with the same feature and also uses the pseudo-channel mode to improve the access granularity without the extra signaling lines required for a true dual channel mode.

On the other hand, memory scheduling algorithms play a significant role in modern memory controllers to improve system performance. Prior works [BI12; Ki10; Su14] proposed different memory scheduling algorithms that comply with various applications and characteristics of specific systems. However, most existing memory controllers employ the commonly used *First Ready First Come First Served scheduler (FR-FCFS)* [Ri00] to shorten the overall latency of DRAM accesses. FR-FCFS was developed to prioritize page hits and address the shortcomings of the *First Come First Served (FCFS)* algorithm [Ri00].

We observe that the GDDR5X pseudo-channel mode provides additional bandwidth that can be utilized to design an effective memory scheduler for improving the system performance. Moreover, GPU DRAM, such as the GDDR family, always can be a good predictor for features that will appear in CPU DRAM in the future. This study, therefore, aims to introduce a novel memory scheduling algorithm to maximize the benefit of using GDDR5X pseudo-channel mode. To this end, we modified gem5 [Bi11] to support GDDR5X as a CPU DRAM and evaluated the performance of our proposed scheduling algorithm using trace-driven simulation.

To the best of our knowledge, this is the first work that investigates the GDDR5X pseudo-channel feature to gain further bandwidth improvement.

The contributions of this paper are as follows:

1. We propose a new buffer scheduling policy called *pseudo-channel-aware scheduling* algorithm to exploit further bandwidth of GDDR5X
2. We evaluate the pseudo-channel-aware scheduling algorithm on a wide variety of workloads and compare its performance to a baseline system. On average, the pseudo-channel-aware scheduling algorithm improves the performance by 38% for an eight-core system across different workloads[SP06].

This paper is organized as follows. Related work is discussed in Section 2. In Section 3 the details of our proposed method is presented. Section 4 discusses the experimental setup. Following that, experimental results are given in Sections 5. Finally, Section 6 presents conclusions and future work.

2 Related Work

DRAM buffer scheduling algorithms have been discussed in several prior works ranging from application-aware to QoS-aware memory schedulers. Rixner et al. [Ri00] proposed the First Ready First Come First Service (FR-FCFS) algorithm. This scheduler picks the first ready access, which is an access from an already open page for the case of the open-page policy. Next, if no access is found, the oldest access from the DRAM read/write buffer is picked. Kim et al. [Ki10] proposed a QoS-aware memory scheduler called *ATLAS* to control the number of services allocated for each application at the memory controllers and to prioritize the requests of applications with respect to the minimum received memory service. Subramanian et al. [Su14] developed an application-aware memory scheduler called *Blacklisting Memory Scheduler*. This scheduler categorizes application requests into two vulnerable-to-interface and interface-causing sets, to improve the performance and fairness with a lower cost.

Although a considerable number of works [BI12; Ki10; Su14] have proposed advanced DRAM scheduling algorithms, many memory controllers utilize the FR-FCFS scheduling algorithm since it presents a good performance [Ha14] compared to other more complex algorithms.

Our literature review revealed none of the existing memory schedulers investigated the GDDR5X pseudo-channel mode to design a pseudo-channel-aware DRAM scheduling algorithm with the goal of improving performance. This study presents the design of such a scheduler to exploit further bandwidth from the pseudo-channel mode.

3 Problem Analysis

In this section, first, we present the GDDR5X standard and explain its most important features. Second, we describe our proposed DRAM scheduling algorithm.

3.1 GDDR5X

GDDR5X SGRAM (Synchronous Graphics Random Access Memory), which is an extension of the well-established GDDR5 standard, employs a new I/O feature to achieve a higher bandwidth compared to GDDR5.

In addition to the regular GDDR5 Dual Data Rate (DDR) mode, a Quad Data Rate (QDR) is also supported and transfers four data words per clock cycle and can be enabled via a mode register bit. The DDR mode operates similar to a GDDR5-SGRAM, and provides a burst size of 8 words. In contrast, the QDR mode transfers a longer burst size of 16 words, which increases the memory access granularity from 32 bytes (in DDR) to 64 bytes. Therefore, GDDR5X-QDR doubles the burst size and memory access granularity respectively.

Using QDR GDDR5X is able to provide additional bandwidth compared to GDDR5. The standard enables 10-14Gb/s/pin [JE16b] data rate which is approximately a 2x increase over its predecessor.

The increase in the access granularity from 32 bytes to 64 bytes in GDDR5X may fetch unnecessary data to the main memory, which can reduce effective memory bandwidth. Therefore, GDDR5X supports a feature called pseudo-channel mode in which the memory is split into two 16-bit pseudo channels. Considering the burst length of 16, in pseudo-channel mode, GDDR5X has the same access granularity as GDDR5. However, these two pseudo channels are not fully independent, which means that while this feature allows issuing two simultaneous memory requests with 32-bytes access granularity, access type, bank, and page needs to match. However, different columns can be selected for each of the two pseudo channels. According to [JE16b], GDDR5X utilizes six column address bits per pseudo channel. Therefore, in pseudo-channel mode, 64 different addresses can potentially be paired for each request. Issuing two simultaneous memory requests in Pseudo-channel mode has no performance penalty. We can thus also think of GDDR5X in pseudo channel mode as a GDDR5, which can issue one additional request for free, if the requirements mentioned above, are met.

3.2 Proposed Memory Scheduler

DRAM buffer scheduling algorithms reorder memory requests to improve memory performance. For example, commonly used FR-FCFS prioritizes (1) the page-hit requests and (2) older requests over the younger ones to boost memory throughput. Given the pseudo-channel feature, it is beneficial to make the memory scheduler aware of this feature in order to maximize DRAM performance.

As remarked in Subsection 3.1, in the case of GDDR5X pseudo-channel mode, the memory controller can fetch an additional request simultaneously without any extra cost if two requests have the same type, bank, page, but different pseudo channels; any two requests which meet these criteria will be referred to as pairable requests in this work.

Tab. 1: Request prioritization in our proposed algorithm

1. **Pairable Requests (open page):** Pairable requests from an already open page are prioritized over all other requests
2. **Pairable Requests (non-yet-open page):** Pairable requests from a non-yet-open page are prioritized
3. **Non-pairable Request (open page):** One open-page-hit request is prioritized
4. **FCFS:** Older request is prioritized over younger ones

Hence, utilizing GDDR5X as a CPU DRAM, we propose an extended FR-FCFS algorithm called pseudo-channel-aware scheduling, to exploit additional bandwidth provided by

the GDDR5X pseudo-channel mode. Table 1 summarizes the four steps of the proposed algorithm.

When enabling pseudo-channel-aware scheduling algorithm, first, the memory scheduler proceeds to select all pairable requests either from the read or write buffer queue. Since opening and closing a DRAM page takes a significant amount of time and energy [Ha14] the memory scheduler first picks all available pairable requests from currently open DRAM pages. In the second step, the pseudo-channel-aware scheduler targets all available pairable requests from DRAM non-yet-open pages. Next, if no more pairable requests have been found, the memory scheduler picks the first available unpaired request from an open DRAM page (step 3). Finally, if no page-hit request is available, the pseudo-channel-aware scheduler operates the same as the FCFS algorithm [Ri00] and targets the oldest request in the queue.

4 Experimental Setup

In this study, in order to quickly assess different policies, we choose a trace-driven approach to evaluate our algorithm. Figure 1 illustrates a modern DRAM controller architecture, with split read and write queues, a memory scheduler, and a response queue. The memory scheduler is responsible for executing memory scheduling policies. We generated memory traces using gem5 and implemented the functionality of a DRAM controller in a custom trace-based simulator, written in C++. This simulator was used to evaluated the effectiveness of our proposed memory scheduling algorithm.

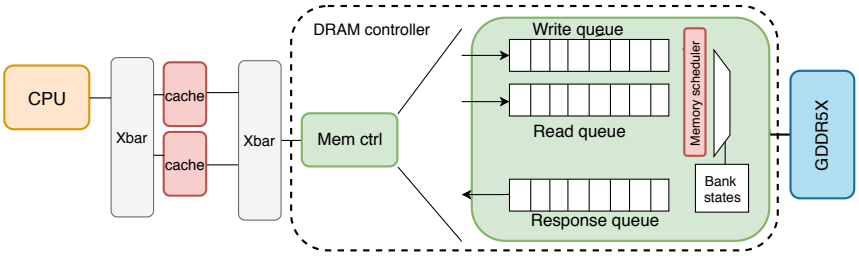


Fig. 1: Overview of DRAM controller architecture

The memory traces were generated with the gem5 simulator using a simulated multi-core system with eight out-of-order cores and a 512 KB shared last level L2 cache. Furthermore, the gem5 simulator was extended in terms of clocking, data rate, and timing parameters [JE16b] to support GDDR5X and its requirements and features. To support the pseudo-channel mode, the address mapping unit in the memory controller was modified to assign a bit to the pseudo channel. For address mapping, we used RoRaBaChPcbCo with Ro, Ra, Ba, Ch, Pcb and Co representing row, rank, bank, channel, pseudo-channel bit and column, respectively, going from MSB to LSB. Table 2 lists the details of all system parameters used in the gem5 simulation.

Tab. 2: Parameters of the gem5 simulation

Processor	Eight-core, 4.0 GHz, Out-of-Order
L1 D/I Cache	32 KB, 2-way set associative
LLC	512 KB, 8-way set associative
Cache Line Size	32 B
Prefetcher at LLC	Stride prefetcher with degree=4
DRAM Controller	On-chip, Open row policy
Address Mapping	RoRaBaChPcbCo
Number of Instructions (sampling)	One billion instructions

In our custom simulator, we utilized the presented pseudo-channel-aware scheduling algorithm to efficiently pick requests from either the read or write queue. As this is a trace-based study, we do not consider read and write dependencies. Therefore, the queues are assumed to be always filled with requests. The simulation was performed with four different queue sizes of 256, 512, 1024, and 2048 entries in both pseudo channel and regular mode. We used workloads from SPEC2006 CPU [SP06]. In particular, we have chosen 14 benchmarks from different categories to evaluate the effectiveness of our proposed memory scheduler. Each benchmark was compiled using GCC and GNU Fortran. All benchmarks are run using the SPEC reference input set.

In order to keep simulation time and trace file size manageable, we used custom gem5 scripts with sampling. As shown in Figure 2, the trace collection function applies sampling to capture all the behavior of the benchmarks and accurately collect DRAM memory accesses.

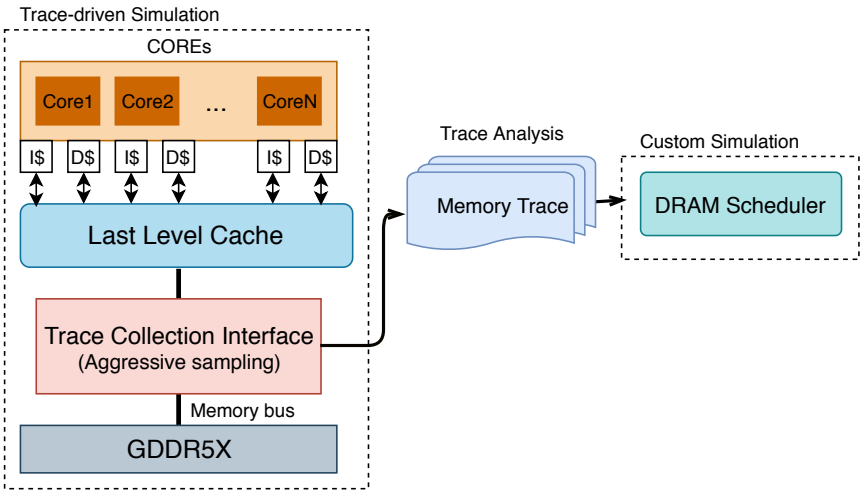


Fig. 2: Overview of our framework

The idea behind sampling is to stay longer in the gem5 functional simulation than in performance simulation and gain significant speedup. Therefore, for each workload, the gem5 CPU was switched back and forth between a O3CPU which is a model of a real out-of-order CPU and a AtomicSimpleCPU [Bi11] which is a model of a purely functional in-order CPU. All simulations were executed using the first one billion instructions.

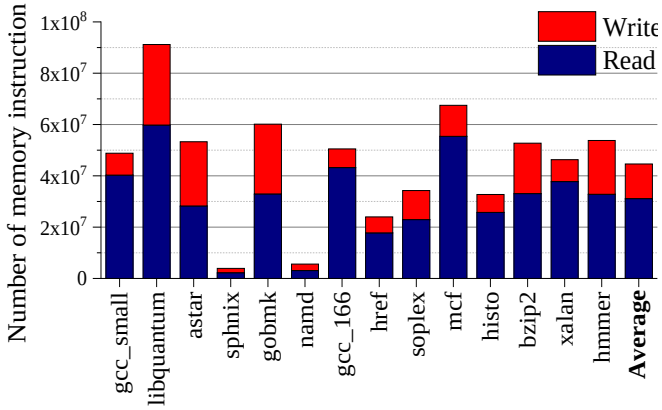


Fig. 3: Memory read/write accesses

5 Experimental Results

As explained in Section 3.1 GDDR5X can operate in two modes: regular and pseudo-channel. Each mode has a different memory access granularity. In the regular mode, GDDR5X uses a large burst size of 64 bytes while in pseudo-channel mode the granularity remains the same as GDDR5 with 32 bytes. Regular mode is the simplest way to use all the bandwidth offered by GDDR5X and was thus chosen as a baseline. However, due to the large access granularity, regular mode will often perform unnecessary reads or writes. As discussed in Section 3.1, pseudo channel mode is more flexible. While both regular and pseudo-channel mode fetch 64 bytes from the same memory page, regular mode basically glues together two 32 byte halves, while in pseudo channel mode, each channel can fetch any 32 byte burst from its half of the memory page. This means each burst on one side of the page can be paired with 64 different bursts from the other half of the page. This additional flexibility of the pseudo-channel mode should increase the effective bandwidth of the DRAM interface and allow us to avoid unnecessary reads or writes.

Figure 3 shows the total number of read/write operations in all benchmarks we studied. The results are extracted from gem5 while running one billion instructions using sampling. As illustrated in Figure 3, the average number of memory accesses in our simulations is approximately five percent of the total number of instructions.

The ratio of total paired requests are presented in Figure 4 and Figure 5 for pseudo-channel and regular modes with limited queue sizes of 256, 512, 1024, and 2048 entries. Moreover, we executed the algorithm with an unlimited queue size for read/write queues to an upper bound for the ratio of paired requests, and thus estimate the maximum achievable improvement in our proposed algorithm. According to Figure 4 and Figure 5, our proposed algorithm yields average upper bounds of 96 percent and 87 percent for the ratio of paired requests in pseudo-channel and regular modes, respectively. Given the limited queue sizes of 256 to 2048, our proposed algorithm on average yields 43 percent (pseudo-channel mode) and 34 percent (regular mode) increase in the number of paired requests. The results show that the proposed algorithm is able to achieve more paired requests in the pseudo-channel mode due its flexibility in selecting paired requests (recall Section 3.1).

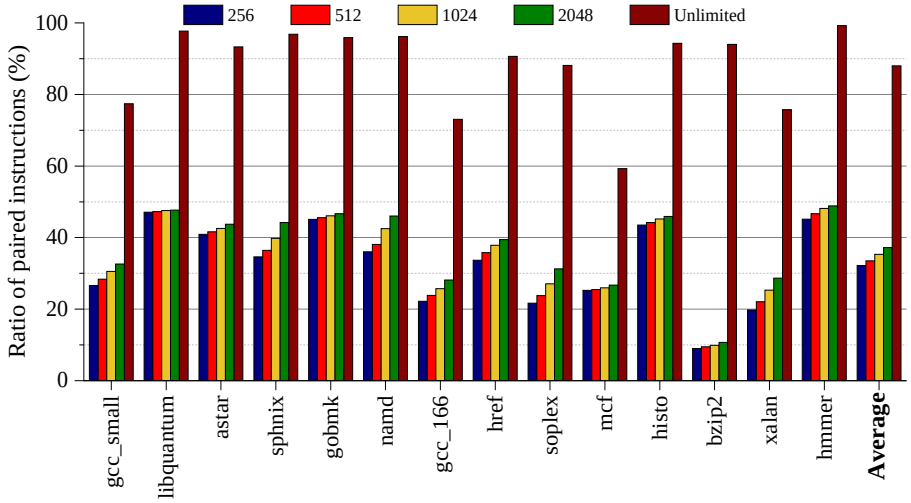


Fig. 4: Number of paired requests over the total number of memory requests in regular mode (baseline)

In order to have a better comparison, the ratio of paired requests is normalized to that of the regular mode of GDDR5X with the smallest queue size of 256 entries. Figure 6 indicates the normalized ratio of total paired requests in pseudo-channel and regular modes across all workloads with different queue sizes. As shown in Figure 6, our algorithm for all four queue sizes in pseudo channel mode can achieve a higher ratio compared to regular mode. In the pseudo-channel mode, the average value of the normalized ratio across different workloads ranges between 1.41 (for the smallest queue size) to 1.64 (for the largest queue size), whereas its average value is limited to less than 1.2 (for the largest queue size) in the regular mode.

Figure 7 shows the percent reduction in total memory accesses obtained in pseudo-channel and regular modes. Similar to Figure 6, the results are normalized to the regular mode of GDDR5X with 256 entries (the smallest queue size). In pseudo-channel mode, Figure 7

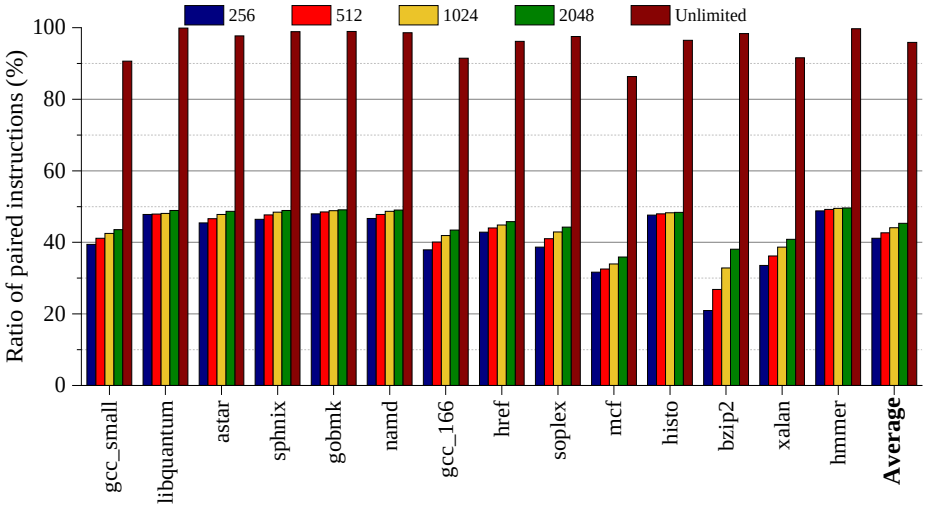


Fig. 5: Number of paired requests over the total number of memory requests in pseudo-channel mode

indicates that pseudo-channel-aware scheduling algorithm on average gives 12.5 percent and 18 percent reductions in number of memory requests for the smallest and largest queue sizes, respectively, for pseudo-channel mode. In contrast, the average reduction in regular mode is bounded to 7.2 percent (in the largest queue size).

As a result, we find that even with the smallest queue size, the proposed algorithm in pseudo-channel mode outperforms the largest queue size in regular mode. This means that our algorithm can achieve higher memory access reduction compared to the baseline across all workloads we studied.

6 Conclusions and Future Work

In this paper, we have presented a novel memory scheduling algorithm called *pseudo-channel-aware scheduling* to exploit the GDDR5X pseudo-channel mode and thus improve memory system performance. The proposed algorithm prioritizes memory requests in read/write queues to select pairable requests with the same accesses type from the same bank and page but different pseudo channels. We have evaluated the proposed scheduling algorithm using 14 various benchmarks from SPEC2006. Furthermore, we performed all simulations with four different read/write queue sizes of 256, 512, 1024, and 2048 entries, with the results been compared to GDDR5X regular mode as a baseline. The results reveal that the proposed algorithm is able to achieve a reduction of 12.5% and 18% on average for the smallest and largest queue sizes of 256 and 2048 entries, respectively. Furthermore,

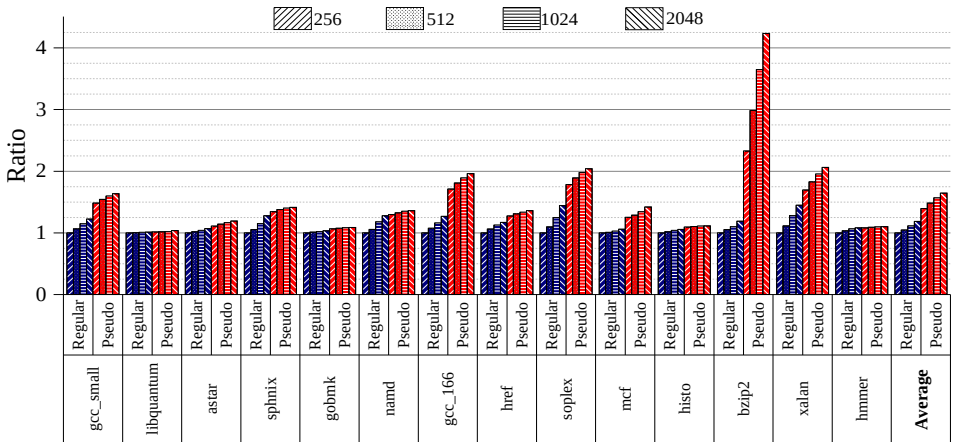


Fig. 6: Ratio of total paired requests normalized to the baseline with a queue size of 256 entries

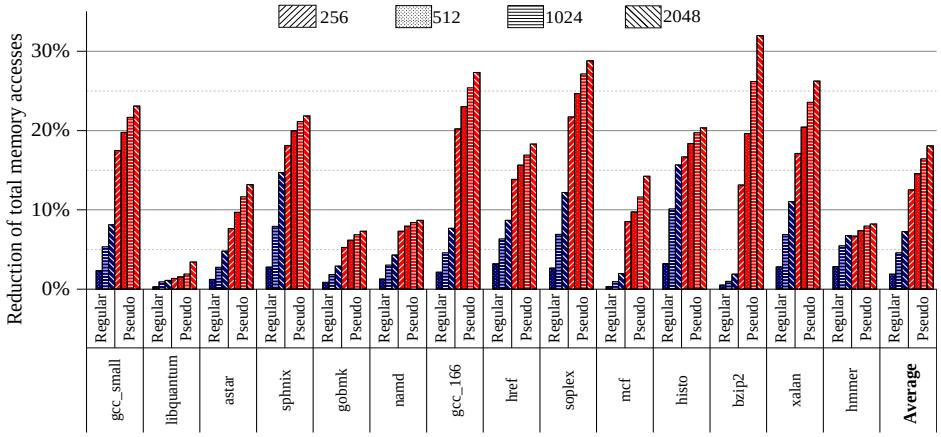


Fig. 7: Percentage of total memory access reduction compared to the baseline with a queue size of 256 entries

our algorithm can achieve an upper bounds of 96% for the ratio of paired requests in pseudo-channel mode, thus demonstrating its potential.

In the future, we aim to continue this study in several directions. As noted in Section 4, this is a trace-based study and read/write dependencies are not reflected in the traces. Therefore, we plan to implement our proposed scheduling policy on gem5 to measure the performance on a real system. In addition, we intend to investigate this study in terms of energy consumption; hence, the effect of the proposed algorithm on energy consumption will be analyzed later. Future work also includes further evaluation with other benchmark suites than SPEC2006 and real-world applications.

References

[Bi11] Binkert, N.; Beckmann, B.; Black, G.; Reinhardt, S. K.; Saidi, A.; Basu, A.; Hestness, J.; Hower, D. R.; Krishna, T.; Sardashti, S., et al.: The Gem5 Simulator. ACM SIGARCH Computer Architecture News 39/02, pp. 1–7, 2011.

[BI12] Bojnordi, M. N.; Ipek, E.: PARDIS: A Programmable Memory Controller for the DDRx Interfacing Standards. ACM SIGARCH Computer Architecture News 40/03, pp. 13–24, 2012.

[Br18] Brox, M.; Balakrishnan, M.; Broschwitz, M.; Chetreanu, C.; Dietrich, S.; Funfrock, F.; Gonzalez, M. A.; Hein, T.; Huber, E.; Lauber, D., et al.: An 8-Gb 12-Gb/s/pin GDDR5X DRAM for Cost-effective High-performance Applications. IEEE Journal of Solid-State Circuits 53/1, pp. 134–143, 2018.

[Ha14] Hansson, A.; Agarwal, N.; Kolli, A.; Wenisch, T.; Udipi, A. N.: Simulating DRAM Controllers for Future System Architecture Exploration. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)., pp. 201–210, 2014.

[JE16a] JEDEC Standard: Graphics Double Data Rate (GDDR5) SGRAM Standard. JESD212C 3/08, 2016.

[JE16b] JEDEC Standard: Graphics Double Data Rate (GDDR5X) SGRAM Standard. JESD232A 5/15, 2016.

[JE17] JEDEC Standard: Graphics Double Data Rate (GDDR6) SGRAM Standard. JESD250 11/17, 2017.

[Ki10] Kim, Y.; Han, D.; Mutlu, O.; Harchol-Balter, M.: ATLAS: A Scalable and High-performance Scheduling Algorithm for Multiple Memory Controllers. In: IEEE 16th International Symposium on High Performance Computer Architecture (HPCA)., pp. 1–12, 2010.

[Ri00] Rixner, S.; Dally, W. J.; Kapasi, U. J.; Mattson, P.; Owens, J. D.: Memory Access Scheduling. In: ACM SIGARCH Computer Architecture News. Vol. 28. 2, pp. 128–138, 2000.

[SP06] SPEC[Online], 2006, URL: https://www.spec.org/.

[Su14] Subramanian, L.; Lee, D.; Seshadri, V.; Rastogi, H.; Mutlu, O.: The Blacklisting Memory Scheduler: Achieving high Performance and Fairness at Low Cost. In: IEEE International Conference on Computer Design (ICCD)., pp. 8–15, 2014.

B. Juurlink, W.Karl (Hrsg.)
Proceedings 28th PARS Workshop

Symptom-based Fault Detection in Modern Computer Systems

Thomas Becker [1], Nico Rudolf[2], Dai Yang[3], Wolfgang Karl [4]

Abstract: Miniaturization and the increasing number of components, which get steadily more complex, lead to a rising failure rate in modern computer systems. Especially soft hardware errors are a major problem because they are usually temporary and therefore hard to detect. As classical fault-tolerance methods are very costly and reduce system efficiency, light-weight methods are needed to increase system reliability. A method that copes with this requirement is symptom-based fault detection. In this work, we evaluate the ability to detect different faults with symptom-based fault detection by using hardware performance counters. As the knowledge of a fault occurrence is usually not enough, we also evaluate the possibility to make conclusions about which fault occurred. For the evaluation, we used the fault-injection library FINJ and manually manipulated loops. The results show that symptom-based fault detection enables the system to detect faulty application behavior, however fine-grained conclusions about the causing fault are hardly possible.

Keywords: System Reliability; Fault Detection; Fault Analysis

1 Motivation

To satisfy the demand of higher computing power, increasing miniaturization of components with increasing complexity and a growing number of components is deployed. This leads to the constant rising of the failure rate of today's computing systems. Especially soft errors in hardware that are random and of temporary nature occur more often, as they are caused by lowering the system voltage in the creation of energy-efficient products [SS02]. These faults are particularly hard to detect as they do not always lead to wrong results and may not be reproducible.

Lightweight methods that increase system reliability are needed as classical methods, like redundancy and checkpoints, increase the cost and significantly reduce the system efficiency [Be08]. One example of a lightweight method is symptom-based fault detection. Symptom-based fault detection identifies faults in the system by comparing values of runtime metrics called symptoms with a database of correct behavior and assuming differing behavior is caused by a fault.

[1] Karlsruher Institut für Technologie, Kaiserstraße 12, 76131 Karlsruhe, Germany thomas.becker@kit.edu
[2] Karlsruher Institut für Technologie, Kaiserstraße 12, 76131 Karlsruhe, Germany nico.rudolf@student.kit.edu
[3] Technische Universität München, Boltzmanstraße 3, 85748 Garching, Germany d.yang@tum.de
[4] Karlsruher Institut für Technologie, Kaiserstraße 12, 76131 Karlsruhe, Germany wolfgang.karl@kit.edu

In this work, we evaluate the ability to detect different faults caused by hardware and system interferences. As the sole knowledge of the occurrence of a fault is not enough to choose an adequate recovery method, we also want to examine if the resulting symptoms allow to make conclusions about the root cause. In summary, we make the following contributions:

- We evaluate the concept of symptom-based fault detection with different faults and system interferences, e.g. created by the fault-injection library FINJ, on a state-of-the-art computing system.
- We analyze if the resulting symptoms make conclusions about the causing fault possible.

The remainder of this paper is structured as follows: Section 2 explains the necessary fundamentals for this work. We describe our method of symptom-based fault detection and how to evaluate it in Section 3. The following Section 4 presents our experimental setup including the fault-injection methods used and our results. We then describe related work in Section 5 and wrap up with conclusions and future work in Section 6.

2 Fundamentals

To define **faults**, **errors** and **failures**, we use the work of Salfner et al. [SLM10]:

- A **failure** refers to misbehavior that can be observed by the user. This means there may be something wrong inside the system, but as long this does not result in incorrect output there is no failure.
- An **error** is defined as the deviation of the system state from the correct state. Hence, an error may lead to the service failure of an system, but also can stay unnoticed.
- **Faults** are then the hypothesized cause of an error. This means that errors are manifestations of faults.

Symptom-based fault detection is based on the following hypothesis: Systems exhibit steady-state performance behavior with few variations in the non-faulty case. However, a fault manifests itself as increasingly unstable performance-related behavior before escalating into a failure [WPN07]. This means that a symptom for occurring faults manifests itsel as a variation of performance-related behavior, which can be monitored by performance counters. To sum up, the basic concept is to monitor performance counters and assume the occurrence of faults if they vary significantly compared to a baseline.

Performance Application Programming Interface (PAPI), developed by the University of Tennessee [Te10], is a user-level library that grants easy access to performance counters. PAPI provides two interfaces for application developers. The high level interface is simple

to use and allows fast access to standard events that are present in most architectures. In contrast, the low level interface allows a more detailed control and the access to so-called native events that are specific to the underlying architecture. In this work, we use the PAPI low level interface.

Fault injection is the deliberate triggering of faults with the objective to observe the resulting behavior and to test error handling code. Fault injection can be done directly in hardware or by using specific software tools. In this work, we focus on software implemented fault injection. As fault injection is an important technique in proving the correctness and robustness of a system or software, many tools and libraries exist. For this work, we chose FINJ [Ne18], a fault injection tool for HPC systems. FINJ is implemented in Python and based upon tasks. Thereby, a task can represent a benchmark or a fault-triggering program. The execution of a workload of tasks is controlled by a specific controller that schedules and starts the tasks on an engine.

3 Method

To evaluate symptom-based fault detection, the first step is to create a database, which stores the performance behavior of correct executions. As a wide range of performance-related metrics are available, relevant metrics have to be filtered out. We define a metric as relevant, if their values do not vary significantly during repeated runs without faults and show significant variance in the presence of faults. Relevant metrics can be found via profiling runs. If the profiling runs only include executions without faults, the set of possibly relevant metrics can be at least reduced to metrics that are stable during repeated executions. Additionally, a lower and an upper threshold for the values of the selected metrics have to be chosen. These thresholds are used to detect anomalies later. If a monitored value lies outside of these thresholds, the occurrence of a fault is suspected.

For a selected number of benchmarks, we execute runs with injected faults. For each run, only one specific fault is used. If there are monitored metrics whose values lie outside of the chosen thresholds, the injected fault is assumed to be detected.

After all experiments are conducted, an analysis step follows. The injected faults are classified. Then, the monitored results are checked if faults belonging to the same class show similar changes in the runtime metrics. We also check if we can differentiate faults belonging to different calsses by observing the monitored runtime behavior. In the current state, this is done by hand, but the process should be automated in the future by using machine learning algorithms.

4 Evaluation

This section presents our fault injection methods, the experimental setup and the conducted experiments. We used three different benchmarks, i.e. matrix multiplications with 300×300

and 500×500 floating-point matrices, as well as Hotspot3D and SRAD of the Rodinia Benchmark Suite. We added PAPI instrumentation code to each benchmark in order to monitor selected performance counters. All experiments are executed ten times with and without fault injection. The results show the average $\varnothing$ and the standard variation s of the ten executions with and without the injected fault. Additionally, we computed an occurrence ratio, that shows how often the value of the performance counter varied significantly from the non-faulty case. This means if a value increases significantly in 8 out of 10 executions, the occurrence ratio would be 80 %.

The experiments are conducted on a server with two Intel Xeon E5-2650 v4 CPUs a 12 cores each and 128 GB with 2400MHz DDR4 SDRAM DIMM (PC4-19200). The software environment includes Ubuntu 18.04.1, the Linux 4.15.0-43-generic kernel and glibc 2.27.

4.1 Fault Injection

In this work, we analyse three types of faults: the alteration of loop index variables to create random memory accesses, the reduction of loop iterations, and interferences created by the FINJ library. The intereferences are used to mimic anomalies in real-life systems by stressing single components, emulating interference or malfunction in that component.

The alteration of loop index variables is done by overwriting the current value of the index variable in the pages of the process in main memory via opening */proc/$procid/mem/* and then jumping to the address of the variable. An example can be seen in Listing 1.

```
FILE *mem = fopen("/proc/$procid/mem/", "w");
fseek(mem, (uintptr_t) &i, SEEK_CUR);
fwrite(&manipulation, sizeof(i), 1, mem);
fclose(mem);
```

List. 1: Altering an index variable

The process id and and the variable adress can be obtained by calling `popen("pid of $processname", "r")` and writing out the adress to a file that can be read by the alteration process, respectively. As we only want to create random accesses, we made sure that the altered index value always lies within the given range and that the correct number of iterations is executed. In the same way, the number of iterations can be altered by overwriting the current iteration bound.

From the FINJ library we used five interference applications: *copy*, *ddot*, *dial*, *leak* ,and *memeater* that are inspired by Tuncer et al. [Tu17].

4.2 Benchmarks

As a first benchmark, we implemented a floating-point matrix multiplication (**mmult**) with 300×300 and 500×500 matrices initialized with random floating-point numbers. Other benchmarks we used are:

Hotspot3D iteratively computes the heat distribution of a 3d chip represented by a grid. In every iteration, a new temperature value depending on the last value, the surrounding values, and a power value is computed for each element. For the evaluation, we used a $512 \times 512 \times 8$ grid with the start values for temperature and power included in the benchmark suite, and a total of 1000 iterations.

SRAD is a diffusion method for ultrasonic and radar imaging applications based on partial differential equations used to remove noise without destroying important image features. The benchmark consists of image extraction, continuous iterations over the image, and image compression. As input, we used the 502×458 image provided by the benchmark suite with 100 iterations and $\lambda = 0.5$.

4.2.1 Experiments

The Alteration of the Loop Index Variable creates random accesses into the used data structure. This changes the data cache behavior of the benchmark increasing misses as the random accesses violate the locality principle. The results of the mMult benchmark (s. Table 1) show these changes. Misses in the data translation lookaside buffer (TLB DM)

Symptom	mMult w/o faults		mMult w faults		
	∅	s	∅	s	occurrence ratio
PAPI TLB DM	27.8	13.8	1882	258.7	100 %
PAPI PRF DM	88898.7	290.1	503561.3	794	100 %
PAPI L2 DCM	15733227	1125.2	16188659.3	4214.8	100 %
PAPI L3 DCA	15733439.3	312.3	16185326.6	1193.8	100 %

Tab. 1: Results of the combination of mmult and the manipulation of the loop index

and in the L2 data cache (L2 DCM), data prefetch misses (PRF DM), and L3 data caches accesses (L3 DCA) increase significantly.

The Reduction of the Number of Iterations effectively leads to a reduction of issued and executed instructions. In general, the instruction performance counters are very precise, e.g. the counters for the executed floating-point operations always match the actually executed operations with a deviation of 0. Therefore, a reduction of the executed instructions is easily recognizable using the provided instruction counters as can be seen exemplary in the results of the mMult benchmark in Table 2. Here, the total floating-point operations, the floating-point additions and floating-point multiplications scale according to the number of

executed loops. As these results are pretty straightforward, we omitted the results of the other two benchmarks.

Symptom	mMult w/o faults	mMult w faults		
	N = 300	N = 200	N = 100	N = 50
PAPI FP OPS	$54 \cdot 10^6$	$36 \cdot 10^6$	$18 \cdot 10^6$	$9 \cdot 10^6$
PAPI FML OPS	$27 \cdot 10^6$	$18 \cdot 10^6$	$9 \cdot 10^6$	$4.5 \cdot 10^6$
PAPI FADD OPS	$27 \cdot 10^6$	$18 \cdot 10^6$	$9 \cdot 10^6$	$4.5 \cdot 10^6$

Tab. 2: Results of the iteration number reduction for the matrix multiplication benchmark

Copy constantly executes file in- and output executions, thereby creating hard drive interferences (I/O overhead). Expected results are variations in the low-level cache structure and the TLB. These expectations are confirmed by the results of the SRAD benchmark shown in Table 3. In 9 out of 10 runs, large variations can be noted for the L3 total cache misses (L3 TCM) and TLB DM counters. Additionally, the total number of stalls increases on average about 3 % and the number of L2 instruction cache misses (ICM) by about 150 % on average. However, these two symptoms occurre less often and in the case of the L2 ICMs vary significantly between runs, which aggravates a detection. The results for mMult are

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurrence ratio
PAPI L3 TCM	1.8	1.87	44.44	54.81	90 %
PAPI TLB DM	11899.7	1080.84	19816.7	3421.12	90 %
STALLS TOTAL	79803752.8	155218.56	82099161.3	1197866.62	80 %
PAPI L2 ICM	2088	154.6	3188.3	1366.39	60 %

Tab. 3: Results of the combination of SRAD and copy

presented in Table 4. There is a similarity to the results of SRAD where the PRF DMs and total stalls increase, but the biggest variations are seen in the instruction caches. Hotspot3D has similar results with an increase in L2 and L3 cache accesses as well as symptoms such as an increase in L3 data cache writes (DCW) and in cycles stalled waiting for memory writes (PAPI MEM WCY).

Symptom	mMult w/o faults		mMult w faults		
	∅	s	∅	s	occurrence ratio
PAPI L2 ICA	92.7	8.06	751.3	29.04	100 %
PAPI L3 ICA	110.1	26.76	227	20.33	100 %
PAPI PRF DM	258091.5	2350.31	270598.5	1349.06	100 %
STALLS TOTAL	33806710.3	448643.96	35368628.1	1226479.31	80 %

Tab. 4: Results of the combination of the mMult and copy

Leak creates a controlled memory leak by constantly allocating new arrays and copying data into them using `memcpy()`. This leads to additional data cache and TLB misses. The results of SRAD in Table 5 show that the number of L3 TCMs and the number of total

cycles increase significantly throughout all test runs. Additional symptoms are TLB DMs and L3 instruction caches accesses (L3 ICA), which are visible in 80 % of the conducted executions. Similar to the copy benchmark, variations for Hotspot3D are mostly visible in

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occ. rat.
PAPI L3 TCM	1.8	1.87	11061.6	17445.73	100 %
PAPI REF CYC	911683861	4293565.15	1042523457.8	18936885.44	100 %
PAPI TLB DM	11899.7	1080.84	16347.5	4986.21	80 %
PAPI L3 ICA	1950.6	101.63	2439.8	262.8	80 %

Tab. 5: Results of the combination of SRAD and leak

the instruction caches (s. Table 6). In this case however, there is no single symptom that is present in all 10 executions. mMult also shows only two symptoms in combination with leak, increases in total stalls and TLB IMs, which are visible in 80 % of the test runs.

Symptom	Hotspot3D w/o faults		Hotspot3D w faults		
	∅	s	∅	s	occurrence ratio
PAPI L2 ICM	1360	269.32	1870.1	252.18	80 %
PAPI L1 ICM	1641.3	251.18	2079.1	358.2	70 %

Tab. 6: Results of the combination of Hotspot3D and leak

Memeater, like leak, creates a controlled memory leak. Additionally, memeater also executes additions, which in total create misses in the instructions caches additional to the symptoms visible for leak. Mirroring the results of leak, the number of total cache misses and the total cycle number increase significantly in all of the test runs for SRAD. TLB data misses are also significantly augmented again and observable in 8 out of the 10 executions. Furthermore, the additional instructions executed then lead to an increase in L2 ICMs. The results for the combination of Hotspot3D and memeater are identical to the combination of Hotspot3D and leak with increases in L1 and L2 ICMs and MEM WCYs. Even the occurrence ratio is identical for all three symptoms. For mMult, the results resemble the results of the copy benchmark instead of leak, as the visible symptoms are increases in L1 and L2 ICMs, total stalls, PRF DMs, and additionally an increase in cycles with maximum instruction issue (FUL ICY).

Dial uses several floating-point math instructions, like `pow()` and `sqrt`, to create interferences in the ALU. As not much additional data is used while performing these operations, mostly variations in the instruction caches are expected. For SRAD, the results are displayed in Table 7. As expected, significant increases in the L2 ICMs are measured. These correlate with the decrease in instructions cache hits (ICH) and an increase in L3 instruction cache accesses (ICA) (and reads (ICR)). All those symptoms are observable in 100 % of the execution runs. The additional data used for the computations lead to an increase in TLB DMs and a decrease in L2 DCAs. Symptoms for mMult are also manifested in the increase in ICMs and correlating increases in L2 and L3 ICAs. Furthermore, the number of prefetch

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurr. ratio
PAPI L2 ICM	2088	154.6	2915.9	162.82	100 %
PAPI L2 ICH	20175.3	512.32	18633	169.35	100 %
PAPI L3 ICA	1950.6	101.63	2956.6	92.42	100 %
PAPI TLB DM	11899.7	1080.84	15885.1	2255.44	100 %
PAPI L2 DCA	19876232.64	2957162.47	10991900.1	14624.57	90 %

Tab. 7: Results of the combination of SRAD and dial

data misses decreases in every run. Hotspot3D also showed significant increases (up to

Symptom	mMult w/o faults		mMult w faults		
	∅	s	∅	s	occurrence ratio
PAPI PRF DM	258091.5	2350.3	236171	15761.98	100 %
PAPI L1 ICM	109.8	16.7	170.5	25.02	90 %
PAPI L2 ICM	130.5	19.34	155.9	14.92	70 %
PAPI L3 ICA	93.4	6.6	130.4	21.98	70 %

Tab. 8: Results of the combination of mMult and dial

400 %) in instruction cache misses. Surprisingly, we measured a decrease in TLB DMs and MEM WCYs.

Ddot is also used to create ALU interferences executing float-point operations. The benchmark allocates and initializes matrices and then executes a floating-point matrix multiplication. Compared to dial, this means that more additional data is used. The results for SRAD (s. Table 9) show the effects of the additional instructions executed on the instruction caches. Again, the ICMs on the L2 level increase, which correlates with the increase of L3 ICAs (and ICRs) and decrease of L2 ICHs. The data usage is not really visible in the monitored values, as the only visible variation was a decrease in L2 data cache accesses, as also seen for SRAD with dial. Similar to the results for dial, we observe

Symptom	SRAD w/o faults		SRAD w faults		
	∅	s	∅	s	occurrence ratio
PAPI L2 ICM	2088	154.6	2784.3	114.36	100 %
PAPI L3 ICA	1950.6	101.63	2830.9	67.07	100 %
PAPI L2 DCA	19517962.78	3226987.33	11026045.4	66239.26	90 %
PAPI L2 ICH	20175.3	512.32	18740.4	445.68	90 %

Tab. 9: Results of the combination of SRAD and ddot

a significant increase in L1 and L2 ICMs for Hotspot3D and also noticed an increase in misses in the instruction TLB (ITLB). Again, the number of TLB data misses decrease compared to the execution without interferences. The mMult benchmark also shows similar results to dial. Increases in ICMs, TLB DMs and a decrease in PRF DMs are again detected. Additionally, we measured increases in ITLB misses and total stalls.

Symptom	Hotspot3D w/o faults		Hotspot3D w faults		
	∅	s	∅	s	occurrence ratio
PAPI L1 ICM	1277.4	208.14	5060.8	286.99	100 %
PAPI L2 ICM	1416.9	249.7	5228.9	268.77	100 %
PAPI TLB DM	513562.6	137417.52	213266.5	4514.72	90 %
ITLB MISS	587.4	257.26	958.2	312.8	70 %

Tab. 10: Results of the combination of Hotspot3D and ddot

4.2.2 Statistical Analysis

To test the statistical significance of our results, we compute the Welch's t-test [WE47], a statistical test that is used to test the hypothesis that two means belong to the same population. If that is the case, the occurring symptom originates from correct behavior and not a fault. Exemplary, the results of three tests are shown here. Tables 11, 12 and 13 show the results for the combination of SRAD and dial, Hotspot3D and leak, and mMult and copy. For all symptoms monitored in these benchmarks, the deviation that occured in

Symptom	df	t	α	t_{crit}	p
PAPI L2 ICM	17.95	-11.66	0.001	-3.922	$8.26 \cdot 10^{-10}$
PAPI L2 ICH	10.94	9.04	0.001	4.437	$2.09 \cdot 10^{-6}$
PAPI L3 ICA	17.84	-23.16	0.001	-3.922	$1 \cdot 10^{-14}$
PAPI TLB DM	12.93	-5.04	0.001	-4.221	$2.31 \cdot 10^{-4}$
PAPI L2 DCA	9.00	9.50	0.001	4.781	$5.47 \cdot 10^{-6}$

Tab. 11: Welch's t-test results for the combination of SRAD and dial

the fault-injection runs has a probabilty to occur in normal runs of less than 1 % and in most cases even less than 0.1 %. This means that it is almost definite that the monitored symptoms are not from the distribution observed in the fault-free runs. Therefore, it is reasonable to say that the injected faults changed the application behavior. For all conducted

Symptom	df	t	α	t_{crit}	p
PAPI L2 ICM	17.92	-4.37	0.001	-3.922	$3.71 \cdot 10^{-4}$
PAPI L2 ICM	16.13	-3.16	0.01	-2.898	0.006

Tab. 12: Welch's t-test results for the combination of Hotspot3D and leak

benchmarks, the maximum probability for a symptom occuring in a fault-free run is 3.6 %. For 18 of 30 symptoms examined, the test resulted in a probability of less than 0.1 %. So in summary, the Welch's t-tests show the statistical significance of the monitored symptoms for all benchmarks in our experiments.

Symptom	df	t	α	t_{crit}	p
PAPI L2 ICA	10.38	-69.1	0.001	-4.437	0
PAPI L3 ICA	16.79	-11	0.001	-3.965	$4.3 \cdot 10^{-9}$
PAPI PRF DM	14.35	-14.59.16	0.001	-4.073	$5.24 \cdot 10^{-10}$
STALLS TOTAL	11.37	-3.78	0.01	-3.012	0.0029

Tab. 13: Welch's t-test results for the combination of mMult and copy

5 Related Work

Symptom-based fault detection has be done before in the literature. Arulaj et al. [Ar13] use performance counters as symptoms to detect concurrency bugs in production-run systems. They access the performance counters via Linux perf.

Williams et al. [WPN07] also use different performance counters to detect anomalous behavior that should form patterns leading up to failures. With an anomaly detector they create a time series that serves as input for a failure predictor. The predictor checks if there is a pattern that indicates escalating instability which then signals an impending failure.

Instead of considering all performance metrics, Narayanasamy et al. [NCC07] focus on the branch predictor, the store set predictor and L2 cache accesses. They assume that a faulty execution leads to an increase in *undesirable outcomes*, e.g. a misprediction by a branch predictor.

The ReStore architecture by Wang et al. [WP05] uses symptom-based fault detection combined with a checkpointing mechanism. If the fault detection signals the occurrence of a fault, the architectural state of an earlier checkpoint is restored. Exceptions, branch mispredictions coupled with a confidence predictor for the branch and event logs that store events, like control instruction outcomes, are used as symptoms.

mSWAT [Ha09] is a fault detection and fault diagnosis framework for multicore architectures. The framework uses a fatal-traps detector, a hang detector checking branch frequencies, a high-OS detector that monitors OS invocations, and a kernel panic detector as symptoms. If a symptom occurs, a diagnosis mechanism is invoked that decides whether the fault is just a software bug or a hardware fault, whether it is a transient or permanent fault and which core is faulty. This is done by tracing and replaying execution.

6 Conclusion

In this work, we evaluated the concept of symptom-based fault detection with three different benchmarks: a matrix multiplication, Hotspot3D and SRAD of the Rodinia Benchmark Suite, combined with seven ways to generate faults and interferences. We generated faults and interferences that affect different parts of a computing system, thereby creating a

classification for the injected faults. We then analyzed the results, if the monitored symptoms allow to conclude which fault originally occurred.

In general, we have found minimally two symptoms for every benchmark fault combination. In the worst case there was at least one symptom with an occurrence ratio of 80 % and for most cases at least one symptom with a ratio of 100 %. We have also shown the statistical significance of the monitored symptoms by computing Welch's t-test. Therefore, it is fair to say that we are able to detect all faults in every benchmark using symptom-based fault detection.

Faults that alter the instruction number are easily detectable as these counters are very precise. A distinction from the other fault classes is also easy, as they do not alter the number of instructions.

Considering each benchmark on their own, the different instances of the interference classes (memory-bound and ALU-bound interferences) have very similar behavior. E.g. SRAD showed significant variations in total L3 cache misses, the number of cycles needed and TLB DMs for all three memory bound benchmarks. However, there is no single set of symptoms that is relevant for every instance of a class over all benchmarks. Only significant variations of instruction cache accesses and misses were visible for each instance of the interference classes over all benchmarks. A possible distinction could be the degree to which the values increase. ALU-bound interferences create larger increases compared to memory-bound ones. Additionally, the memory-bound interferences create more variations in data related counters in most cases. Differentiating between the interferences and the loop index manipulation is possible as the loop index manipulation does not affect the instruction caches. However, to be able to make a differentiation the complete set of symptoms has to occur simultaneously. For five symptoms with an occurence ratio of 80 % each, the probabilty that all symptoms occur together is only around 33 %. Without the whole set of symptoms however, a useful differentiation is not possible. As a summary, we conclude that symptom-based fault detection is very useful to detect faulty application behavior. Coarse-grained conclusions about the causing fault are generally possible, but finer distinctions need additional tool support.

In the future, we plan to extend the evaluation to GPUs. Additionally, to make our approach practically usable, we will integrate it into a library-based runtime system designed for user support of heterogeneous architectures, thereby automating the instrumentation process, the search for relevant performance metrics and the analysis of the results.

References

[Ar13] Arulraj, J.; Chang, P.-C.; Jin, G.; Lu, S.: Production-run Software Failure Diagnosis via Hardware Performance Counters. SIGARCH Comput. Archit. News 41/1, pp. 101–112, Mar. 2013, ISSN: 0163-5964, URL: http://doi.acm.org/10.1145/2490301.2451128.

[Be08] Bergman, K.; Borkar, S.; Campbell, D.; Carlson, W.; Dally, W.; Denneau, M.; Franzon, P.; Harrod, W.; Hiller, J.; Karp, S.; Keckler S. an d Klein, D.; Lucas, R.; Richards, M.; Scarpelli, A.; Scott, S.; Snavely, A.; Sterling, T.; Williams, R. S.; Yelick, K.; Kogge, P.: ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems Peter Kogge, Editor & Study Lead, 2008.

[Ha09] Hari, S. K. S.; Li, M.; Ramachandran, P.; Choi, B.; Adve, S. V.: mSWAT: Low-cost hardware fault detection and diagnosis for multicore systems. In: 2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). Pp. 122–132, Dec. 2009.

[NCC07] Narayanasamy, S.; Coskun, A. K.; Calder, B.: Transient Fault Prediction Based on Anomalies in Processor Events. In: 2007 Design, Automation Test in Europe Conference Exhibition. Pp. 1–6, Apr. 2007.

[Ne18] Netti, A.; Kiziltan, Z.; Babaoglu, Ö.; Sîrbu, A.; Bartolini, A.; Borghesi, A.: FINJ: A Fault Injection Tool for HPC Systems. CoRR abs/1807.10056/, 2018, arXiv: 1807.10056, URL: http://arxiv.org/abs/1807.10056.

[SLM10] Salfner, F.; Lenk, M.; Malek, M.: A Survey of Online Failure Prediction Methods. ACM Comput. Surv. 42/3, 10:1–10:42, Mar. 2010, ISSN: 0360-0300, URL: http://doi.acm.org/10.1145/1670679.1670680.

[SS02] Schiffmann, W.; Schmitz, R.: Technische Informatik 2. Springer Berlin Heidelberg, 2002.

[Te10] Terpstra, D.; Jagode, H.; You, H.; Dongarra, J.: Collecting Performance Data with PAPI-C. In (Müller, M. S.; Resch, M. M.; Schulz, A.; Nagel, W. E., eds.): Tools for High Performance Computing 2009. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 157–173, 2010, ISBN: 978-3-642-11261-4.

[Tu17] Tuncer, O.; Ates, E.; Zhang, Y.; Turk, A.; Brandt, J.; Leung, V. J.; Egele, M.; Coskun, A. K.: Diagnosing Performance Variations in HPC Applications Using Machine Learning. In (Kunkel, J. M.; Yokota, R.; Balaji, P.; Keyes, D., eds.): High Performance Computing. Springer International Publishing, Cham, pp. 355–373, 2017, ISBN: 978-3-319-58667-0.

[WE47] WELCH, B. L.: THE GENERALIZATION OF 'STUDENT'S' PROBLEM WHEN SEVERAL DIFFERENT POPULATION VARLANCES ARE INVOLVED. Biometrika 34/1-2, pp. 28–35, Jan. 1947, ISSN: 0006-3444, eprint: http://oup.prod.sis.lan/biomet/article-pdf/34/1-2/28/553093/34-1-2-28.pdf, URL: https://doi.org/10.1093/biomet/34.1-2.28.

[WP05] Wang, N. J.; Patel, S. J.: ReStore: symptom based soft error detection in microprocessors. In: 2005 International Conference on Dependable Systems and Networks (DSN'05). Pp. 30–39, June 2005.

[WPN07] Williams, A. W.; Pertet, S. M.; Narasimhan, P.: Tiresias: Black-Box Failure Prediction in Distributed Systems. In: 2007 IEEE International Parallel and Distributed Processing Symposium. Pp. 1–8, Mar. 2007.

B. Juurlink, W.Karl (Hrsg.)
Proceedings 28th PARS Workshop

Weight Pruning for Deep Neural Networks on GPUs

Thomas Hartenstein, Daniel Maier, Biagio Cosenza, Ben Juurlink
Embedded Systems Architecture, Technische Universität Berlin, Germany
thomas.hartenstein@campus.tu-berlin.de, {daniel.maier,cosenza,b.juurlink}@tu-berlin.de

Abstract: Neural networks are getting more complex than ever before, leading to resource-demanding training processes that have been the target of optimization. With embedded real-time applications such as traffic identification in self-driving cars relying on neural networks, the inference latency is becoming more important. The size of the model has been identified as an important target of optimization, as smaller networks also require less computations for inference. A way to shrink a network in size is to remove small weights: weight pruning. This technique has been exploited in a number of ways and has shown to be able to significantly lower the number of weights, while maintaining a very close accuracy compared to the original network. However, current pruning techniques require the removal of up to 90% of the weights, requiring high amount of redundancy in the original network, to be able to speedup the inference as sparse data structures induce overhead. We propose a novel technique for the selection of the weights to be pruned. Our technique is specifically designed to take the architecture of GPUs into account. By selecting the weights to be removed in adjacent groups that are aligned to the memory architecture, we are able to fully exploit the memory bandwidth. Our results show that with the same amount of weights removed, our technique is able to speedup a neural network by a factor of 1.57× given a pruning rate of 90% while maintaining the same accuracy when compared to state-of-the-art pruning techniques.

Keywords: deep neural network; pruning; GPUs; optimization

1 Introduction

Contemporary AI applications are often build using deep neural networks (DNNs). DNNs have improved over the last years becoming state-of-the-art not only for the majority computer vision algorithms but also they have been shown to give superior results in numerous other applications like speech recognition, natural language processing or the discovery of new drugs. In many of these applications, hard real-time deadlines have to be met in order to ensure user satisfaction or even prevent disastrous outcomes, e.g., for self-driving cars. However, to achieve better accuracy, the networks have become also more complex and the network sizes have grown significantly: While the AlexNet Caffemodel is over 200 MB in size, the improved VGG-16 Caffeemodel has already grown to more than 500 MB [HMD15]. More complex networks are composed of more layers and layers have become bigger, leading to resource-demanding training processes that have been the target of optimization in the past. However, for embedded real-time applications (e.g., traffic identification and object detection in self-driving cars) relying on neural networks, the

inference latency is more important. The size of a model has been identified as an important target of optimization as the size is directly related to the number of operations necessary for inference.

Research has shown that models contain a considerable amount of redundancy [De13]. Many connections in the neural network that represent the weights have no or only a minor role when deriving the result. These weights can be removed without affecting the accuracy of network significantly [HMD15; LDS90].

The optimization process of removing weights from a neural network is called *weight pruning*. The general concept of weight pruning is shown in Figure 1. All weights below a certain threshold, in this example 0.3, are removed from the network. Using this approach we can learn the important connections in the network. The result is a new network that contains only the relevant connections of the original network while connections with a negligible influence have been removed. Weight pruning is able to improve the memory usage, as less weights need to be stored in memory. Furthermore, the number of operations needed to compute the result of the network is reduced. The number of weights directly translates to the memory usage and is also closely related to the number of operations needed.

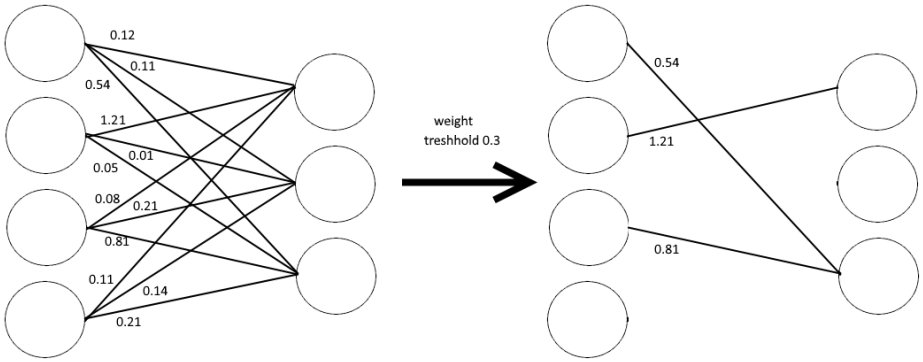


Fig. 1: Weight pruning removes weights below a certain threshold from a neural network.

We propose to use a new technique for weight pruning that overcomes limitations of the state-of-the-art pruning for GPUs [Yu17]. Memory-aware weight pruning is able to accelerate the inference time of deep neural networks by removing weights in continuous groups of multiple weights. These groups are optimized to match the memory architecture of GPUs.

In particular, we make the following contributions: 1. a novel weight pruning technique for neural networks on GPUs; and 2.an evaluation of our technique in terms of inference time and accuracy of the network.

This paper is organized as follows: In Section 2 the related work is introduced and we relate our work to the state-of-the-art. Section 3 introduces our technique for pruning of weights. We describe the experimental setup in Section 4 and show the results in Section 5. Finally, we conclude our work in Section 6.

2 Related Work

Neural networks contain a significant amount of redundant information. Therefore, the computational and memory requirements can both be optimized without a loss in accuracy [De13].

Redundancy in neural networks can be reduced using different techniques. One approach is quantization. By using less bits to store the weights of the network, the overall storage requirements are lowered. Gupta et al. use 16 bit fixed-point number representation for for the calculations of their neural network. [Gu15]. Gong et al. [Go18] show that a pre-trained neural network can be quantized to 8-bit without the necessity of having to retrain the network.

Another popular technique is pruning. Pruning is the removal of filters, weights or whole neurons from a network. The conceptual idea of removing weights is quite old [LDS90]. Pruning can be implemented in a variety of ways: One method is to remove complete filters from Convolutional Neural Networks [Hu18; Li17; Mo17]. Learning the important connections by first removing weights and then retraining the network was first shown by [Ha15]. Yu et al. [Yu17] show that by exploiting the Compressed Sparse Row format on single-instruction-multiple-data (SIMD) units of a microcontroller, pruning can be implemented by removing connections between two neurons. However, the authors learned that weight pruning on GPUs actually slows down the inference time. This deceleration is attributed to the overhead due to sparse data structures which can only be overcome by a pruning rate of 97%. However, pruning rates of more than 90% have shown to lead to a strong decrease in accuracy [Ha15; Yu17] and, therefore, weight pruning was not implemented on GPUs.

3 Weight Pruning for Deep Neural Networks on GPUs

In this work, we optimize weight pruning for the use on GPUs. GPUs have a very distinctive memory architecture, where accesses to the global memory have a high latency and the memory width is wide (e.g., 128 byte). The latency can be hidden by the massively parallel architecture of GPUs. The wide memory architecture connects the density of the information in global memory with the efficiency of memory bandwidth utilisation: In a sparse data layout where 1 out of 16 bytes is requested from memory, the bandwidth utilisation is the same as for a request of 16 bytes. Additionally, the memory architecture requires threads to access the memory in a coalesced manner.

Our approach for memory-aware weight pruning takes the distinct memory architecture of GPUs into account. Instead of pruning individual weights without considering the memory architecture, we propose to treat the weights in groups: First, the weights are organized in groups of adjacent weights with a configurable size. Then, all weights in a group are aggregated and then evaluated. All groups with an aggregated value below a defined threshold are removed from the network. The threshold is determined by the aggregated values of the groups in order to achieve a target pruning rate. Afterwards, the remaining groups are transferred to the compressed sparse row format (CSR). We use this format to be able to use sparse matrix computations in order to accelerate the computations. Sparse matrix-matrix multiplications are optimized to exploit matrices where only a small number of values is different from zero. However, the sparse formats come with some overhead.

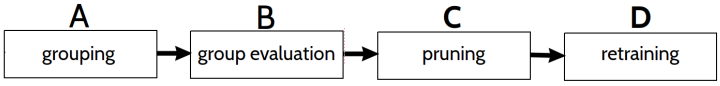


Fig. 2: The subsequent steps of our Memory-aware Weight Pruning technique.

An overview of our technique is depicted in Figure 2. First, we arrange all weights in groups according to the selected group size g in step (A). Then we evaluate the aggregated weight of each group in step (B). In step (C) we perform the actual removal of weights. First, we calculate the threshold t necessary to achieve a given pruning rate and then we remove all weights of all groups with a smaller aggregated weight. Finally, the network is retrained in step (D).

The advantage of using adjacent groups of weights is that they can be loaded at the same time in memory. This ensures that the available memory bandwidth to global memory is used efficiently. In order to be able to evaluate the significance of a group of weights we need to aggregate the weights in the group. We use the RMS (root mean square) function to calculate the aggregated weight of a group, in the same way as related work (e.g., [Yu17]). The motivation is that a high value of a weight has a strong influence on the activation of the neuron in the next layer and that high values will be further amplified by the RMS aggregation of weights.

However, we evaluated different weight aggregation functions (root mean square, arithmetic mean, median, random) and we were not able to observe significant differences in in terms of their influence on the final accuracy of the retrained network. Therefore, we assume that the selection of the groups is not as critical as it might look but the pruning rate and the retraining dictate the accuracy.

The weight matrix stored in CSR format is multiplied with a dense matrix or vector. The weight groups are selected consecutivly within a row of the matrix and with an offset that is a multiple of the group size. We show an example of the grouping step A in Equation 1. Weight matrix M is a 4×4 matrix. The round brackets indicate the weight groups with a

group size $g = 2$. The matrix contains the weights w which form the groups G. The group dimension of the matrix M is 4x2. $G_{1,1}$ is the group consisting of the of weights $w_{1,1}$ and $w_{1,2}$.

$$\mathrm{M} = \begin{bmatrix} \begin{pmatrix} w_{1,1} & w_{1,2} \end{pmatrix} \begin{pmatrix} w_{1,3} & w_{1,4} \end{pmatrix} \\ \begin{pmatrix} w_{2,1} & w_{2,2} \end{pmatrix} \begin{pmatrix} w_{2,3} & w_{2,4} \end{pmatrix} \\ \begin{pmatrix} w_{3,1} & w_{3,2} \end{pmatrix} \begin{pmatrix} w_{3,3} & w_{3,4} \end{pmatrix} \\ \begin{pmatrix} w_{4,1} & w_{4,2} \end{pmatrix} \begin{pmatrix} w_{4,3} & w_{4,4} \end{pmatrix} \end{bmatrix} = \begin{bmatrix} G_{1,1} & G_{1,2} \\ G_{2,1} & G_{2,2} \\ G_{3,1} & G_{3,2} \\ G_{4,1} & G_{4,2} \end{bmatrix} \tag{1}$$

In step B the aggregated weight of each group is calculated.

$$\mathrm{RMS} = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2} \qquad \text{and} \qquad M_{RMS} = \begin{bmatrix} \mathrm{RMS}(G_{1,1}) & \mathrm{RMS}(G_{1,2}) \\ \mathrm{RMS}(G_{2,1}) & \mathrm{RMS}(G_{2,2}) \\ \mathrm{RMS}(G_{3,1}) & \mathrm{RMS}(G_{3,2}) \\ \mathrm{RMS}(G_{4,1}) & \mathrm{RMS}(G_{4,2}) \end{bmatrix} \tag{2}$$

Next, a threshold is defined. The aggregated weights of the groups are sorted by value to select groups with the smallest influence (smallest absolute value). Depending on the pruning rate the threshold is selected. The groups which are below this threshold value are set to 0. In our example, after the groups have been set to 0, the matrix has the following form:

$$M_{prun} = \begin{bmatrix} \begin{pmatrix} 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 \end{pmatrix} \\ \begin{pmatrix} w_{2,1} & w_{2,2} \end{pmatrix} & \begin{pmatrix} w_{2,3} & w_{2,4} \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 \end{pmatrix} \\ \begin{pmatrix} 0 & 0 \end{pmatrix} & \begin{pmatrix} w_{4,3} & w_{4,4} \end{pmatrix} \end{bmatrix} \tag{3}$$

In step D the matrix M_{prun} is converted to the CSR format. The matrix takes the following form:

$$\mathrm{A} = \begin{bmatrix} w_{2,1} & w_{2,2} & w_{2,3} & w_{2,4} & w_{4,3} & w_{4,4} \end{bmatrix}$$

$$\mathrm{JA} = \begin{bmatrix} 0 & 1 & 2 & 3 & 2 & 3 \end{bmatrix}$$

$$\mathrm{IA} = \begin{bmatrix} 0 & 0 & 4 & 4 & 6 \end{bmatrix}$$

The new representation of the matrix consists of A, JA and IA. It is CSR format and contains only the weights of the neural network that are greater than zero. These are the weights that were selected in step B and C and will be used later.

Tab. 1: Details of our system used to conduct the results

Hardware	Model
CPU	Intel Core i5-7200U
GPU	NVIDIA Geforce GTX 950M
GPU main memory	2 GB
Software	**Version**
Ubuntu	16.04.5 LTS
CUDA	9.0.176
Python	3.5.2
Keras	2.2.4
TensorFlow	1.10.1

4 Experimental Evaluation

In this section we briefly introduce our experimental evaluation. First, we examine how to measure the execution time, then we explain how we measure the accuracy of the neural network.

4.1 Performance

As our pruning technique is optimized for fully connected layers and the inference of fully connected layers is based on matrix-matrix multiplications we conduct our performance evaluation using a matrix-matrix multiplications benchmark. All our experiments are performed using an NVIDIA Geforce GTX 950M GPU. We execute the matrix-matrix multiplication calculations in CSR format on the graphics card. The conventional calculation of matrix-matrix multiplications is called dense matrix-matrix multiplication below, and we use NVIDIA's implementation for these multiplications [Nv12].

To calculate the matrix-matrix multiplication on the GPU and to measure the execution time we use CUDA 9[Nv18]. The sparse matrix-matrix multiplications are performed with the NVIDIA's cuSPARSE library. The dense matrix-matrix multiplications are performed with the library cuBLAS.

In our benchmark two matrices of dimensions 4096×4096 and 4096×50 are multiplied with each other. The matrix sizes are chosen to achieve comparability with the work of Yu et al.[Yu17] The pruned weights are merged into the summarized format described in Section 3. For each of our performance experiments we measure the kernel execution time only, as this share of the overall execution time is the predominant part.

cuSparse offers the CSR and HYB sparse formats for calculations. For the CSR format, the library offers a matrix-matrix multiplication where the first matrix is a sparse matrix and the second matrix is a dense matrix. The HYB format is a mix of ELL format and COO format. Unfortunately, for HYB format, cuSparse offers no support for matrix-matrix multiplication but only a function for a matrix vector multiplication. For this reason, the CSR format was chosen. In the calculation of the inference of a fully connected neural network, above all, the matrix-matrix multiplication is the predominant part of work load. The addition of bias has, according to our investigations, only a minor role. We do not evaluate our pruning technique in terms of training time. The total training time of the network is increased, because the retraining time of the pruned network is added to the training time of the network.

4.2 Accuracy

In this section we describe how the accuracy of the network was determined. We use the MNIST dataset [LC19] that consists of 60,000 images of handwritten digits. Each image has a size of 28x28 pixels and can belong to 1 of 10 categories spanning the numbers between 0 and 9. The data set is randomly separated into two distinct subsets: 1) training data (54 000 images) and 2) test data for validation (6 000 images) in order to avoid over-fitting.

Tab. 2: Structure of our neural network

Layer part	Layer type	Activation function	Size
input	fully connected	ReLU	784
hidden	fully connected	ReLU	128
hidden	fully connected	ReLU	128
hidden	fully connected	ReLU	256
hidden	fully connected	ReLU	256
hidden	fully connected	ReLU	512
hidden	fully connected	ReLU	512
output	fully connected	Softmax	10

To evaluate our technique we use a neural network that consists of eight fully connected layers. This network serves the purpose to allow us to assess our technique. The structure of our network is shown in Table 2. Each of the 784 ($= 28 \times 28$) pixels represents an input of the input layer. We use the stochastic gradient descent (SGD) as optimization function.

First, we have to train the network and therefore we train the model for 3000 epochs using the training data set before we start the pruning process. In order to study the effects of the group size the weight matrices were subdivided into different group sizes during the pruning process and then the root-mean-square (RMS) is determined for each group. We implement our technique using the Keras API with the TensorFlow backend. When the aggregated weight of the groups is determined the groups are sorted by the aggregated weight. Then we set the threshold in a way such that the given pruning rate is reached. Finally, the aggregated

weights with an RMS smaller than the threshold are removed. After pruning, we retrain the neural network over 6000 epochs. The number of training epochs before and after training are chosen to match the related work [Yu17].

4.3 Performance

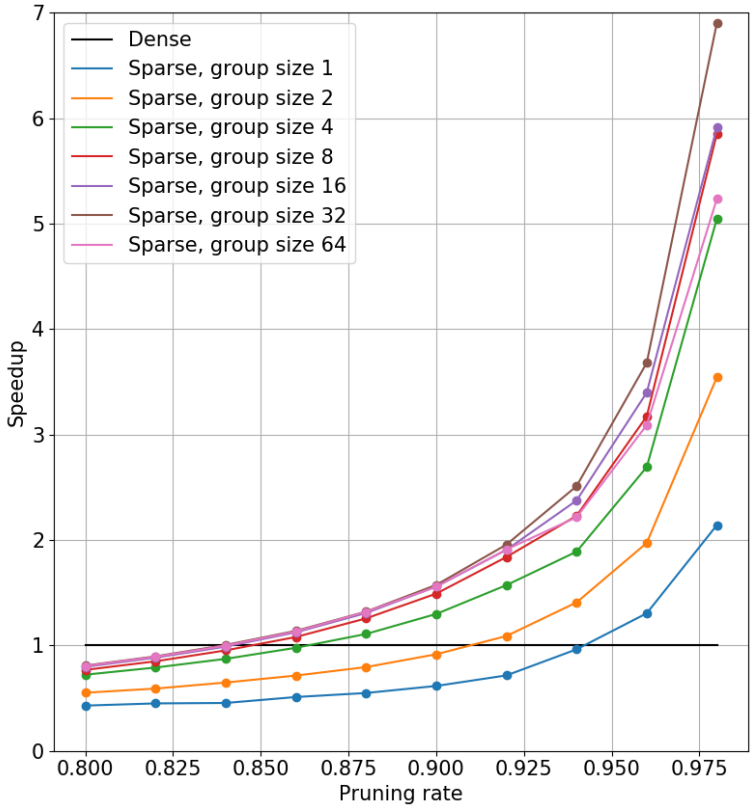


Fig. 3: Speedup of sparse matrix-matrix multiplication compared to dense matrix-matrix multiplication. The matrices have the sizes of 4096x4096 and 4096x50.

5 Results

In this section, we discuss the results of our experiments. Figure 3 shows how the performance is affected when using different pruning rates. Figure 4 shows the accuracy of different group sizes when setting the pruning rate to 90%. In Figure 5 the first two results are related to each other.

Figure 3 shows the speedup of a sparse matrix-matrix multiplication of size 4096×4096 by 4096×50 for different pruning rates between 80% and 98% and group sizes of 1, 2, 4, 8, 16, 32 and 64. The speedup is calculated by normalizing the execution time of the pruned network to the execution time of the dense network. We reproduce the work of Yu et al. and show in their results labeled *sparse, group size 1*. In our benchmark the sparse matrix-matrix multiplication at a pruning rate of 94% without grouping was as fast as the dense matrix-matrix multiplication. A similar pruning rate without grouping was reported by Yu et al. considered too large, because the accuracy would drop too much. We show that for a sparse matrix-matrix multiplication with a group size of 32, the speedup is greater than 1 at a pruning rate of 84% compared to a pruning rate of 94% for the state-of-the-art of Yu et al. In their work, the lowest reported pruning rate of 90% required more than twice the time when compared to conventional dense matrix-matrix multiplication. When we apply our grouped pruning technique, we reach a speedup of 57% achieved given the same pruning rate.

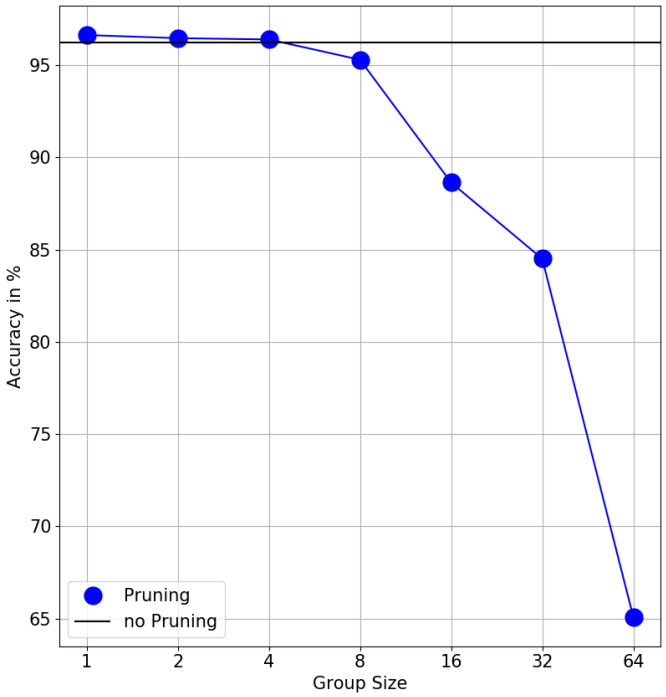


Fig. 4: Achieved accuracy per group size of the pruning technique. The pruning rate set to 90%.

In Figure 3 we show that a higher pruning rate leads to a higher speedup in the sparse matrix-matrix multiplications. The highest speedup is achieved with a group size of 32. We can attribute the speedup at least partially to the amount of coalesced memory accesses. Coalesced memory accesses are important on GPUs in order to exploit the memory bandwidth. The *Global Load Efficiency* (as reported by NVProf) increases from 67 % to

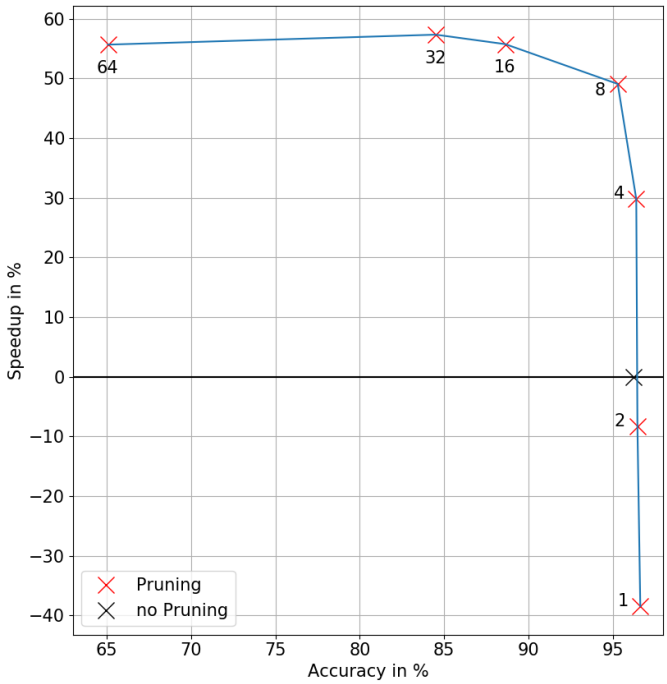


Fig. 5: Achieved accuracy in relation to the achieved speedup for different group sizes when setting the pruning rate to 90%.

82 % for a group size of 1 if the pruning rate is increased from 80 % to 98 %. From a group size of 8 the *Global Load Efficiency* rises to over 99 %.

5.1 Accuracy

In this section we discuss the accuracy achieved with different group sizes at 90% pruning rate. Figure 4 shows the accuracy of the network shown in Table 2. The horizontal line shows the accuracy of accuracy comparable to trained network that was not pruned. The blue dots show the accuracy of the networks, which were pruned with a pruning rate of 90%. The figure shows that although the pruning rate was set to 90% for all networks, the accuracy decreases the larger the group size. A sharp decline in accuracy can be observed as the group size increases. With smaller group sizes, the accuracy of the network could be maintained or even slightly improved. We assume that the improvement is the consequence of an increased training time of the network. The accuracy is higher for a small group size and many groups than for a large group size and fewer groups, as it is more likely to prune the weights that do not contribute to the activation of the neuron.

Figure 5 shows the accuracy values of the Figure 4 in relation to the GPU execution times of sparse matrix-matrix multiplications at 90% pruning rate in Figure 3. The black cross in the figure denotes the neural network without pruning. The red crosses mark speedup and accuracy of the neural network when we apply our technique and set the pruning rate to 90%. The group sizes 1, 2, 4, 8, 16, 32 and 64 of the pruning networks are written next to the respective cross of the result. It can be seen that a group size of 32, as already shown in Figure 5, offers the highest speed gain, but severely limits the accuracy of the network. Group sizes 1 and 2 even cause the neural network to perform the inference slower because the overhead generated by the CSR format is greater than the speed gain produced by exploiting Global Load Efficiency of the GPU. When aiming for an accuracy comparable to the original dense network, a group size of 8 is superior, because at this size the accuracy of the neural network is 95.30% while the speedup of 49.08% over the dense network. Sizes 1, 2 and 4 have a significantly lower speedup at a pruning rate of 90 %, as shown in the Figure 3, since the locality of the weights in the memory can not be used here.

6 Conclusion

In this paper, we propose to use memory-aware weight pruning for accelerating the inference time of deep neural networks on GPUs. Our techniques remove weights in a fully-connected layer in continuous groups of multiple weights. By aligning the weight groups to match the size of the memory architecture of current GPUs, we are able to accelerate the inference time by a factor of 1.5× for a given pruning rate of 90%. Furthermore, by using our technique we are able to lower the required pruning rate necessary to be profitable on a GPU to 84%, while state-of-the-art pruning requires a pruning rate as high as 94%. We explore how the group size affects the accuracy and what group size is optimal when given a target accuracy. In future work we will investigate different ways of determining the ranking of the weight groups, as our observation that even randomly selected weight groups result in an equal accurate network is very interesting. We will explore the design space given by different matrix sizes in the matrix-matrix multiplications. Additionally, we will be researching domain-specific sparse matrix representations in order to exploit the distinct properties of sparse neural networks. We plan to study the effects of our approach on different networks, new GPU generations and more complex applications.

References

[De13] Denil, M.; Shakibi, B.; Dinh, L.; De Freitas, N., et al.: Predicting parameters in deep learning. In: Advances in neural information processing systems. Pp. 2148–2156, 2013.

[Go18] Gong, J.; Shen, H.; Zhang, G.; Liu, X.; Li, S.; Jin, G.; Maheshwari, N.; Fomenko, E.; Segal, E.: Highly Efficient 8-bit Low Precision Inference of Convolutional Neural Networks with IntelCaffe. eprint arXiv:1805.08691v1/, 2018.

[Gu15] Gupta, S.; Agrawal, A.; Gopalakrishnan, K.; Narayanan, P.: Deep learning with limited numerical precision. In: International Conference on Machine Learning. Pp. 1737–1746, 2015.

[Ha15] Han, S.; Pool, J.; Tran, J.; Dally, W.: Learning both weights and connections for efficient neural network. In: Advances in neural information processing systems. Pp. 1135–1143, 2015.

[HMD15] Han, S.; Mao, H.; Dally, W. J.: Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149/, 2015.

[Hu18] Huang, Q.; Zhou, K.; You, S.; Neumann, U.: Learning to Prune Filters in Convolutional Neural Networks. eprint arXiv:1801.07365v1/, 2018.

[LC19] LeCun, Y.; Cortes, C.: MNIST handwritten digit database./, 2019, URL: http://yann.lecun.com/exdb/mnist/.

[LDS90] LeCun, Y.; Denker, J. S.; Solla, S. A.: Optimal brain damage. In: Advances in neural information processing systems. Pp. 598–605, 1990.

[Li17] Li, H.; Kadav, A.; Durdanovic, I.; Samet, H.; Graf, H. P.: Pruning Filters for efficient ConvNets. ICLR/, 2017.

[Mo17] Molchanov, P.; Tyree, S.; Karras, T.; Aila, T.; Kautz, J.: Pruning Convolutional Neural Networks for resource efficient inference. ICLR/, 2017.

[Nv12] Nvidia: CUDA Toolkit 4.2 CUSPARSE Library. PG05329041v01/, 2012.

[Nv18] Nvidia: Parallele Berechnungen mit CUDA, (WWW), 2018, URL: https://www.nvidia.de/object/cuda-parallel-computing-de.html.

[Yu17] Yu, J.; Lukefahr, A.; Palframan, D.; Dasika, G.; Das, R.; Mahlke, S.: Scalpel: Customizing DNN Pruning to the Underlying Hardware Parallelism. ISCA/, 2017.

B. Juurlink, W. Karl (Hrsg.)
Proceedings 28th PARS Workshop

GPU-beschleunigte Time Warping-Distanzen

Jörg P. Bachmann[1] Kevin M. Trogant[2] Johann-C. Freytag[3]

Abstract: Immer mehr Algorithmen konnten durch Implementierung auf GPUs um mehrere Größenordnungen beschleunigt werden. Insbesondere existieren hochparallele Implementierungen des im Bereich der Zeitreihenanalyse weit verbreiteten Algorithmus' Dynamic Time Warping (`DTW`). Dieser Algorithmus berechnet einen Ähnlichkeitswert zweier Zeitreihen (z. B. Temperaturverläufe) unter Berücksichtigung zeitlicher Variationen wie z. B. zeitliche Verschiebungen. Leider können die existierenden GPU-Implementierungen von `DTW` nicht beliebige zeitliche Variationen berücksichtigen.

In dieser Arbeit stellen wir Implementierungen für GPUs vor, die dieser Einschränkung nicht unterliegen. In unserer Evaluierung zeigen wir, dass sie einen Geschwindigkeitsvorteil von ca. zwei Größenordnungen gegenüber einer CPU-Implementierung erreichen.

Keywords: Dynamic Time Warping; GPU; Algorithmen

1 Einleitung

Viele Algorithmen konnten durch Parallelisierung auf moderner Hardware (z. B. GPUs) erfolgreich beschleunigt werden. Dazu gehören u. a. Sortieralgorithmen [Ar17], Algorithmen aus der linearen Algebra und dem Machine Learning [Ch14; TDB10] und selbstverständlich unzählige Algorithmen aus der Computergrafik.

Zahlreiche Beispiele von Adaptionen der Algorithmen auf die GPU kommen aus der Ähnlichkeitssuche, in der aus einer großen Menge von Objekten (der *Kandidatenmenge*) diejenigen gesucht werden, die einem *Anfrageobjekt* ähnlich sind: Bereits mit älterer Hardware aus 2009 konnte der für Sequenzalignment häufig verwendete Smith-Waterman Algorithmus um etwa eine Größenordnung beschleunigt werden [SA09]; Das im Bereich der Ähnlichkeitssuche häufig verwendete Dynamic Time Warping (`DTW`) [SC90] konnte durch GPUs um ein bis zwei Größenordnungen beschleunigt werden [HSS14; Sa10]; Durch Implementierung des R-Baumes [Be90] auf die GPU konnten die Ausführungszeiten von Ähnlichkeitssuchen um ca. eine Größenordnung beschleunigt werden [YZG13].

`DTW` berechnet ein Maß für die Ähnlichkeit (bzw. Distanz) zweier Zeitreihen, das robust gegen zeitliche Verzerrungen ist. Existierende Implementierungen des Algorithmus' beschränken die möglichen zeitlichen Verzerrungen, um die Berechnung zu beschleunigen [HSS14] oder

[1] Joerg.Bachmann@informatik.hu-berlin.de
[2] Kevin.Trogant@informatik.hu-berlin.de
[3] Freytag@informatik.hu-berlin.de

spezialisieren sich auf die Suche von Subsequenzen ausschließlich fester Länge innerhalb einer langen Zeitreihe [Sa10].

In dieser Arbeit stellen wir neue Implementierungen von DTW für GPUs vor, die von keiner der eben genannten Einschränkungen betroffen sind. Beginnend mit einer Variante für Zeitreihen beschränkter Länge führen wir anschließend Varianten für Sub- und Supersequenzsuche sowie die Möglichkeit, beliebig lange Zeitreihen zu verarbeiten, ein. Abschließend evaluieren wir die Implementierungen ausführlich und zeigen, dass mit Hilfe der GPU Geschwindigkeitsvorteile gegenüber einer CPU-Implementierung von ca. zwei Größenordnungen erreicht werden können. Um den Vergleich nachvollziehbar und möglichst unabhängig von der konkreten CPU zu gestalten, haben wir uns für eine sequentielle Implementierung auf der CPU entschieden. Zwar lassen sich Zeitreihen nicht unter DTW mittels metrischer Indexstrukturen [BKL06; CPZ97; NB09] indizieren, doch wir haben mit dieser Arbeit eine Grundlage geschaffen, Zeitreihen unter DK[4] [Fr06] mittels metrischer Indexstrukturen GPU-beschleunigt zu indizieren.

In Kapitel 2 stellen wir die DTW- und DK-Distanzfunktion ausführlich vor. Die Implementierungen werden in Kapitel 3 beschrieben und in Kapitel 4 evaluiert. Wir schließen mit einem kurzen Fazit in Kapitel 5 ab.

2 Time Warping Distanzfunkionen

Dynamic time warping (DTW) ist eine Funktion, die zwei Zeitreihen einen Abstandswert zuordnet [SC90]. Sie wurde ursprünglich im Bereich der Spracherkennung eingeführt und findet noch heute weite Verbreitung in der Analyse von Zeitreihen. Die Fréchet-Distanz (DK) ist eine Alternative zu DTW, welche die Dreiecksungleichung erfüllt[5].

Einfach formuliert versucht DTW (bzw. DK), den Abstand der Trajektorien (also die Menge der besuchten Punkte im Raum) zweier Kurven zu messen. Betrachtet man zum Beispiel den Weg zweier Personen durch eine Stadt entlang der selben Route (Trajektorie), so gibt es möglicherweise zeitliche Unterschiede durch unterschiedliches Lauftempo und unterschiedlicher Ampelschaltungen. Während sich die Kurven also stark unterscheiden können (wenn man die Punkte der Wege zu jeweils gleichen Zeiten vergleicht), können die Trajektorien identisch sein.

DTW ist für zwei Zeitreihen $S = (s_1, \cdots, s_m)$ und $T = (t_1, \cdots, t_n)$ wie folgt rekursiv definiert:

$$\begin{aligned} \texttt{DTW}(S, ()) &= \infty \\ \texttt{DTW}((), T) &= \infty \\ \texttt{DTW}((s), (t)) &= d(s,t)^2 \end{aligned} \qquad \texttt{DTW}(S,T) = d(s_1,t_1)^2 + \min \begin{cases} \texttt{DTW}(\texttt{Tail}(S), \texttt{Tail}(T)) \\ \texttt{DTW}(S, \texttt{Tail}(T)) \\ \texttt{DTW}(\texttt{Tail}(S), T) \end{cases}$$

[4] DK ist eine Alternative zu DTW, welche die Dreiecksungleichung erfüllt, vollständig invariant unter zeitlichen Verzerrungen ist und sehr ähnlich zu DTW implementiert wird [BF17].

[5] Dadurch eignet sich die DK-Distanz zur Indizierung von Zeitreihen mittels metrischer Indexstrukturen [BKL06; CPZ97; NB09]

wobei $\texttt{Tail}(T) := (t_2, \cdots, t_n)$. Die Abstandsfunktion $d(s,t)$ wird vom Nutzer definiert und entspricht im einfachsten Fall dem Betrag der Differenz zweier Zahlen. Die DK-Distanz ändert sich nur in der Art der Kombination des rekursiven Ergebnisses mit dem aktuellen Abstand beider Zeitreihenelemente:

$$\begin{aligned} \texttt{DK}(S, ()) &= \infty \\ \texttt{DK}((), T) &= \infty \\ \texttt{DK}((s), (t)) &= d(s,t)^2 \end{aligned} \qquad \texttt{DK}(S,T) = \max\left(d(s_1,t_1)^2, \min\left\{ \begin{matrix} \texttt{DK}(\texttt{Tail}(S), \texttt{Tail}(T)) \\ \texttt{DK}(S, \texttt{Tail}(T)) \\ \texttt{DK}(\texttt{Tail}(S), T) \end{matrix} \right\} \right)$$

Da beide Funktionen so ähnlich in der Berechnung sind, erläutern wir im Rest dieses Kapitels nur DTW. Alle Aussagen gelten analog für die DK-Distanz.

Algorithmen, die DTW berechnen, werden üblicherweise mittels dynamischer Programmierung implementiert [SC90]. Abbildung 1 zeigt eine Beispielrechnung. Es wird das Kreuzprodukt beider Zeitreihen S und T gebildet, sodass eine Matrix entsteht, welche die paarweisen Abstände $\|s_i - t_j\|$ der einzelnen Elemente der Zeitreihen enthält. Die Matrix wird so angeordnet, dass sich die Zelle mit dem Abstand $\|s_1 - t_1\|$ links unten befindet.

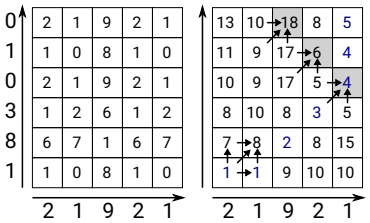


Abb. 1: Beispielmatrizen während der Berechnung von DTW (links das Kreuzprodukt der am Rand stehenden Zeitreihen und rechts die DTW-Matrix mit einem fett markierten optimalen Pfad). Die Pfeile repräsentieren exemplarisch die Abhängigkeiten der Berechnung der Zellen.

Anschließend wird ein Pfad von links unten nach rechts oben gesucht, der die Summe im Fall der besuchten Einträge minimiert. Dabei muss der Pfad zusammenhängend sein und darf niemals nach links oder unten gehen, d. h. innerhalb des Pfades muss der Nachfolger der Zelle mit den Indizes (i, j) entweder $(i+1, j)$, $(i, j+1)$ oder $(i+1, j+1)$ sein. Die Summe der Zellen entlang eines solchen *minimalen* Pfades ergibt das Ergebnis der DTW-Funktion.

Zur Berechnung eines minimalen Pfades wird jede Zelle durch die Summe ihres Inhalts und der kleinsten Vorgängerzelle ersetzt. Um eine Zelle zu ersetzen, müssen die Vorgängerzellen bereits nach diesem Schema angepasst worden sein, sodass eine Abhängigkeit von links unten nach rechts oben entsteht. Es ist leicht zu sehen, dass dieser Algorithmus eine quadratische Komplexität hat (genauer $m \times n$, wenn m und n die Längen der Zeitreihen sind). Mehr noch wurde gezeigt, dass es keine Algorithmen geben kann, die eine bessere Komplexität haben [BK15; Br14]. Algorithmus 1 stellt den Pseudo-Code für eine auf

dynamischer Programmierung basierende Implementierung zur Verfügung. Im folgenden Kapitel 3 implementieren wir den Algorithmus auf der GPU.

Algorithmus 1 CPU-Implementierung von DTW

```
1  Eingabe: Zeitreihen A und B; Ausgabe: Distanz DTW(A, B)
2  M := m × n Matrix
3  for x = 1 to n
4    for y = 1 to m
5      if x = 1 and y = 1 then M_{1,1} = |A_1 − B_1|
6      else if x = 1 and 1 < y then M_{y,1} = M_{y−1,1} + |A_y − B_1|
7      else if x ≤ n and y = 1 then M_{1,x} = M_{1,x−1} + |A_1 − B_x|
8      else if x ≤ n and 1 < y then
9        M_{y,x} = min{M_{y−1,x}, M_{y,x−1}, M_{y−1,x−1}} + |A_y − B_x|
10 return M_{m,n}
```

3 Implementierung

In diesem Kapitel stellen wir verschiedene Implementierungen für die Berechnung von DTW bzw. DK vor. Da sich die Funktionen nur in einer Operation unterscheiden (DK wählt das Maximum zweier Werte, die von DTW addiert werden), erläutern wir nur die Implementierung von DTW.

Die in Kapitel 3.1 vorgestellte Implementierung liefert die Basis für die weiteren Implementierungen. Sie unterstützt nur die Eingabe kleiner Zeitreihen, da diese vollständig in den lokalen Speicher der GPU kopiert werden. In Kapitel 3.2 zeigen wir, wie die Implementierung so abgeändert werden kann, dass sie die Sub- und Supersequenzsuche unterstützt. In Kapitel 3.3 erweitern wir die Basis-Implementierung, sodass eine der beiden Eingabezeitreihen beliebig lang sein kann. Diese Implementierung ist orthogonal mit der Sub- bzw. Supersequenz-Implementierung kombinierbar, sodass ein Algorithmus für GPU-beschleunigte Sub- bzw. Supersequenzsuche mit DTW bzw. DK zur Verfügung gestellt wird. Aus Platzgründen gehen wir darauf nicht weiter ein.

3.1 Basis-Implementierung

Aus Abbildung 1 ist zu erkennen, dass die Berechnung der Zellen innnerhalb einer Diagonalen unabhängig voneinander sind. Zerlegt man die DTW-Matrix in solche Diagonalen von links unten nach rechts oben beginnend (vgl. Abb. 2), so ist die Berechnung einer Diagonalen nur von den Werten der beiden Vorgängerdiagonalen abhängig. Basierend auf dieser Erkenntnis berechnen alle hier vorgestellten GPU-Implementierungen die DTW-Matrix sukzessive von links unten nach rechts oben, halten stets nur die letzten drei Diagonalen im lokalen Speicher und parallelisieren innerhalb der aktuell zu berechnenden Diagonalen.

Algorithmus 2 enthält den relevanten Ausschnitt des CUDA C Codes für die Berechnung von DTW[6].

Algorithmus 2 Basis Implementierung

```
1  // t: Kandidat; q: Anfrage; jeweils vom globalen in den lokalen Speicher kopiert
2  // c: Aktuelle−, p; Letzte−, p2: Vorletzte Diagonale; jeweils im lokalen Speicher
3  // R: Rückgabewert im globalen Speicher
4  // TL: Kandidatenlänge, QL: Anfragelänge,
5  if (threadIdx.x == 0) p2[0] = 0.0f;
6  for (int i = 0; i < QL + TL - 1; ++i) {
7    const int x = threadIdx.x, y = i - x;
8    if (y>=0 && y<TL) c[x+1] = abs(t[y]-q[x])+min(p[x+1],p[x],p2[x]);
9    if (threadIdx.x == 0) p2[0] = INFINITY;
10   __syncthreads();
11   volatile float *_t = p2; p2 = p; p = c; c = _t;
12 }
13 if (threadIdx.x == 0) R[blockIdx.x] = p[QL];
```

Da die Zugriffszeiten auf den lokalen Speicher um Größenordnungen geringer sind als auf den globalen Speicher, kopiert unsere Implementierung beide Zeitreihen zunächst in den schnellen lokalen Speicher (Variablen t und q) und berechnet die Diagonalen der DTW-Matrix ebenfalls im lokalen Speicher (Variablen c, p und p2). Ferner werden Diagonalen in Arrays abgelegt, sodass die Threads jegliche Speicherzugriffe ohne Verzögerung durch Bankkonflikte durchführen.

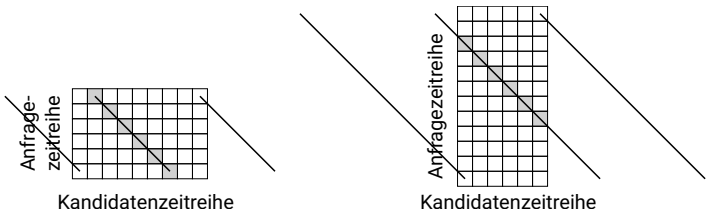


Abb. 2: Skizze zur Basis-Implementierung von DTW: Diagonale Linien repräsentieren Threads, die auf den Zellen der DTW-Matrix arbeiten.

Die maximale Länge der Diagonalen wird durch die Länge der kürzeren Zeitreihe bestimmt. Um jedoch den Programmcode innerhalb der Schleife (Zeile 6 bis 12) möglichst einfach, instruktionsarm und damit schnell zu gestalten, setzen wir die Länge einer Diagonalen mit der Länge der Anfragezeitreihe gleich (vgl. Abbildung 2). Diese Herangehensweise ist effizient, falls die Anfragezeitreihe kürzer ist, als der zu vergleichende Kandidat. Falls die Anfragezeitreihe länger ist, so reserviert die Implementierung für jede Diagonale mehr

[6] Eine vollständige Implementierung ist hier zu finden: http://www.dbis.informatik.hu-berlin.de/fileadmin/projects/GPUAlgorithms/dtw_on_gpus.tar.gz

Speicher, als notwendig ist. Da DTW jedoch symmetrisch bzgl. der Eingabeparameter ist, können im zweiten Fall die Parameter einfach vertauscht werden, um wiederum auf den ersten effizienten Fall zurückzugreifen.

Wie bereits erwähnt, sind die Zellen innerhalb einer Diagonalen unabhängig voneinander, sodass die Länge der Diagonalen gleichzeitig den Grad der Parallelität innerhalb eines Blocks angeben. Darüber hinaus kann durch die Anzahl der zu berechnenden DTW-Instanzen (also die Anzahl der CUDA-Blöcke) parallelisiert werden. Die hohe Geschwindigkeit unserer Implementierung resultiert also aus dem Arbeiten im lokalen Speicher sowie der maximal möglichen Parallelisierung bei minimaler und bankkonfliktfreier Speicherverwaltung.

3.2 Sub- und Supersequenzanfragen

Um auch Sub- und Supersequenzanfragen beantworten zu können, verfolgen wir den Ansatz von S-DTW [AF13; Mü07], wobei im Gegensatz zu DTW der minimale Pfad in einer beliebigen Zelle am linken Rand der DTW-Matrix startet und (rechts davon) in einer beliebigen Zelle des oberen Randes der DTW-Matrix endet (siehe Abbildung 3).

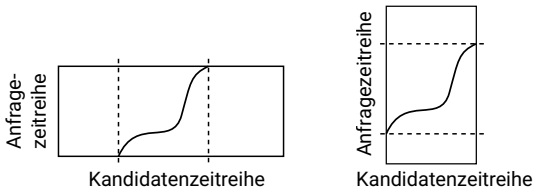


Abb. 3: Skizze der DTW-Matrix zur Berechnung von S-DTW (links: Sub-, rechts: Supersequenzsuche).

Die Implementierung ändert sich dazu nur geringfügig: Jede Zelle der untersten Zeile der DTW-Matrix wird behandelt, als wäre sie die Zelle der linken unteren Ecke in der Basis-Implementierung, d. h. ihre Berechnung ist nicht mehr abhängig von ihrem linken Nachbarn. Das Ergebnis der Berechnung entspricht dem kleinsten Wert der obersten Zeile statt des Wertes der Zelle der rechten oberen Ecke. Analog dazu wird in der Supersequenzsuche ein minimaler Pfad von linken Spalte zur rechten Spalte berechnet.

3.3 Implementierung für lange Zeitreihen

Die Größe des lokalen Speichers bestimmt die maximale Länge der verarbeiteten Zeitreihen bei der Basis-Implementierung. Wir ändern die Implementierung derart, dass kurze Anfragezeitreihen gegen beliebig lange Kandidatenzeitreihen verglichen werden können.

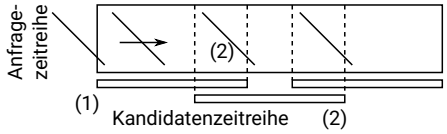


Abb. 4: Skizze der DTW-Matrix bei langen Kandidatenzeitreihen (oben); der überlappenden Fenster (unten); und der berechnenden Threads (Diagonalen oben).

Abbildung 4 skizziert den Ablauf: Wir laden stets nur ein Fenster der langen Kandidatenzeitreihe in den lokalen Speicher (1). Sobald die zu berechnende Diagonale Daten benötigt, die nicht im aktuellen Fenster liegen, wird das nächste Fenster geladen (2). Auf diese Weise laden wir die Zeitreihe sukzessiv in sich überlappende Fenster in den lokalen Speicher.

Um die Daten des überlappenden Bereichs nicht mehrfach aus dem globalen Speicher zu kopieren, verschieben wir sie innerhalb des Puffers im lokalen Speicher. Beim Verschieben der Daten können Race Conditions beim Lesen bzw. Schreiben verschiedener Threads auf die selbe Speicherstelle auftreten. Diese Konflikte treten nicht auf, sobald das Fenster mindestens doppelt so lang wie der überlappende Bereich (d. h. der Länge der Anfragezeitreihe) ist. Wir verlangen im Folgenden solche Fenstergrößen, um die Kosten entsprechender Synchronisierungsbefehle einzusparen.

4 Evaluierung

In diesem Kapitel evaluieren wir die in Kapitel 3 vorgestellten Implementierungen von `DTW` auf der GPU und zeigen, dass unsere GPU-Implementierungen bis zu zwei Größenordnungen schneller sind, als ihr CPU-Pendant. Da sich die Laufzeiten der Sub- und Supersequenzvarianten kaum messbar von der Basisimplementierung unterschieden, verzichten wir hier auf eine Präsentation der Ergebnisse.

In unseren Experimenten haben wir alle GPU-Algorithmen auf einer NVIDIA GeForce GTX 980 Ti ausgeführt. Der CPU-Algorithmus wurde auf einem AMD Ryzen 7 1700X Prozessor ausgeführt. Vereinzelte Experimente wurden auf einer NVIDIA GeForce GTX 780 Ti ausgeführt.

Datensatzeigenschaften Da die Anzahl der Rechenoperationen in den Algorithmen offensichtlich unabhängig von den konkreten Inhalten der Zeitreihen ist, haben wir ausschließlich synthetische Daten benutzt. Die einzigen Größen, von denen die Laufzeit abhängig ist, sind die Länge der Zeitreihen, die Fenstergröße bei der GPU-Implementierung für lange Zeitreihen sowie die Größe des Datensatzes. Da die GPU-Implementierungen von `DTW` kein symmetrisches Verhalten bezüglich der Längen beider Eingabezeitreihen haben, untersuchen wir den Einfluss beider Parameter getrennt voneinander.

Beschreibung der Experimente Innerhalb eines Experiments berechnen wir N verschiedene Abstände mittels `DTW` auf jeweils zufällig erzeugte Zeitreihen A und B vorgegebener Längen $\#A$ und $\#B$. Wir berechnen die Werte von `DTW` mit jeder vorgestellten Implementierung und messen jeweils die kumulierten Laufzeiten. Abbildung 5 zeigt repräsentative Beispielmessungen für die Laufzeiten der Implementierungen bei variierender Länge der Anfragezeitreihe (links), Kandidatenzeitreihe (mitte) sowie bei steigender Anzahl der zu berechnenden `DTW`-Instanzen (rechts).

Analyse der Messwerte Abbildung 5 zeigt, dass die GPU-Implementierungen in fast allen Fällen ca. zwei Größenordnungen schneller sind, als ihr CPU-Pendant. Dagegen zeigt das Diagramm rechts in der Abbildung, dass die CPU bei der Berechnung von nur einer

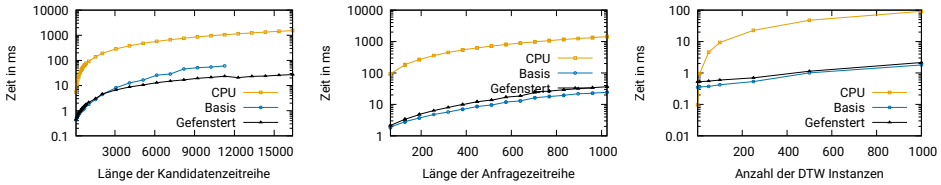


Abb. 5: Laufzeit von DTW. bei variierender Länge der Kandidatenzeitreihe (links); Länge der Anfragezeitreihe (mitte); Anzahl der DTW-Instanzen (rechts). Konstante Paramter: Anzahl DTW-Instanzen: 1000; Länge der gepufferten Zeitreihen: 1024; Länge der ungepufferten Zeitreihen: 64; Pufferlänge: doppelte Länge der Anfragezeitreihe

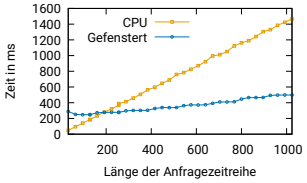


Abb. 6: Laufzeit von DTW. bei variierender Länge der Anfragezeitreihe. Anzahl DTW-Instanzen: 1; Länge der Kandidatenzeitreihe: 1024000;Pufferlänge: doppelte Länge der Anfragezeitreihe

DTW-Instanz schneller ist, als die GPU. Steigt die Anzahl der zu berechnenden DTW-Instanzen jedoch, so profitiert die GPU-Implementierung von der Parallelisierung der Berechnungen.

Abbildung 6 zeigt, dass das gleiche Phänomen bei höherer Parallelisierung einzelner DTW-Instanzen zu beobachten ist: Der Grad der Parallelisierung ist durch die Länge der Diagonalen, d. h. laut Algorithmus 2 durch die Länge der Anfragezeitreihe bestimmt. Ferner beobachten wir, dass die Laufzeit der GPU-Implementierung bei wachsender Länge der Anfragezeitreihe sprunghaft ansteigt. Wir vermuten, dass dieser Effekt auf Schedulingentscheidungen innerhalb der GPU zurückzuführen ist.

Abbildung 5 (links) zeigt, dass die gepufferte Implementierung wider Erwarten schneller als die Basis-Implementierung ist. Wir vermuten den Grund dafür in der Konkurrenz verschiedener DTW-Instanzen um den lokalen Speicher: Durch den wachsenden lokalen Speicherverbrauch der Basis-Implementierung (bei wachsender Länge der Anfragezeitreihe) können weniger DTW-Instanzen parallel ausgeführt werden, während die gepufferte Implementierung gegen diesen Effekt immun ist. Weitere Experimente haben diese Vermutung bestätigt: Abbildung 7 zeigt, dass die gefensterte Implementierung langsamer wird, je größer wir die Fensterlänge wählen. Darüber hinaus beobachten wir, dass die modernere Hardware (NVIDIA GeForce GTX 980 Ti) gegen diesen Effekt robuster als ihr Vorgängermodell (NVIDIA GeForce GTX 780 Ti) ist.

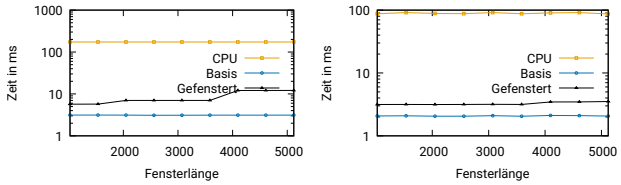


Abb. 7: Laufzeit von `DTW`. bei variierender Pufferlänge. Anzahl `DTW`-Instanzen: 500; Länge der Kandidatenzeitreihe: 256; Länge der Anfragezeitreihe: 512; Berechnung mit einer NVIDIA GeForce GTX 780 Ti (links) und einer 980 Ti (rechts)

5 Fazit

In dieser Arbeit haben wir speichereffiziente GPU-Implementierungen von `DTW` bzw. `DK` inklusive Versionen für Sub- und Supersequenzsuche vorgestellt. Wir haben die Implementierungen um die Möglichkeit erweitert, beliebig lange Kandidatenzeitreihen verabeiten zu können. Unsere Evaluierung ergab einen stabilen Geschwindigkeitsvorteil von ein bis zwei Größenordnungen gegenüber einer CPU-Implementierung. Beobachtungen zufolge wird der Geschwindigkeitsgewinn nicht durch Speicherzugriffe beschränkt, sondern durch den Parallelitätsgrad, den wir durch Optimierung des lokalen Speicherverbrauchs maximieren konnten. Der Flaschenhals wird folglich durch die Größe des lokalen Speichers bestimmt.

Damit wurde eine Grundlage geschaffen, metrische Indexstrukturen mittels GPU-beschleunigter Abstandsmaße zu verbessern. Eine mögliche Anwendung ist, die in dieser Arbeit vorgestellten Implementierungen für andere auf dynamischer Programmierung basierende Algorithmen zu adaptieren (z. B. Levenshtein-Distanz, Hidden-Markov-Chains).

Literatur

[AF13] Anguera, X.; Ferrarons, M.: Memory efficient subsequence DTW for Query-by-Example Spoken Term Detection. In: 2013 IEEE ICME. S. 1–6, Juli 2013.

[Ar17] Arkhipov, D. I.; Wu, D.; Li, K.; Regan, A. C.: Sorting with GPUs: A Survey. CoRR abs/1709.02520/, 2017, arXiv: 1709.02520.

[Be90] Beckmann, N.; Kriegel, H.-P.; Schneider, R.; Seeger, B.: The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. SIGMOD Rec. 19/2, S. 322–331, Mai 1990, ISSN: 0163-5808.

[BF17] Bachmann, J. P.; Freytag, J.-C.: Dynamic Time Warping and the (Windowed) Dog-Keeper Distance. In (Beecks, C.; Borutta, F.; Kröger, P.; Seidl, T., Hrsg.): Similarity Search and Applications. Springer International Publishing, Cham, S. 127–140, 2017, ISBN: 978-3-319-68474-1.

[BK15] Bringmann, K.; Künnemann, M.: Quadratic Conditional Lower Bounds for String Problems and Dynamic Time Warping. CoRR abs/1502.01063/, 2015.

[BKL06] Beygelzimer, A.; Kakade, S.; Langford, J.: Cover trees for nearest neighbor. In: Proceedings of the 23rd ICML. ICML '06, ACM, Pittsburgh, Pennsylvania, S. 97–104, 2006, ISBN: 1-59593-383-2.

[Br14] Bringmann, K.: Why walking the dog takes time: Frechet distance has no strongly subquadratic algorithms unless SETH fails. CoRR abs/1404.1448/, 2014.

[Ch14] Chetlur, S.; Woolley, C.; Vandermersch, P.; Cohen, J.; Tran, J.; Catanzaro, B.; Shelhamer, E.: cuDNN: Efficient Primitives for Deep Learning. CoRR abs/1410.0759/, 2014, arXiv: 1410.0759.

[CPZ97] Ciaccia, P.; Patella, M.; Zezula, P.: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In: Proceedings of the 23rd International Conference on Very Large Databases. VLDB '97, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, S. 426–435, 1997, ISBN: 1-55860-470-7.

[Fr06] Fréchet, M. R.: Sur quelques points du calcul fonctionnel. Rendiconti del Circolo Mathematico di Palermo, 1906.

[HSS14] Hundt, C.; Schmidt, B.; Schömer, E.: CUDA-Accelerated Alignment of Subsequences in Streamed Time Series Data. In: 2014 43rd ICPP. S. 10–19, Sep. 2014.

[Mü07] Müller, M.: Information Retrieval for Music and Motion. Springer-Verlag, Berlin, Heidelberg, 2007.

[NB09] Novak, D.; Batko, M.: Metric Index: An Efficient and Scalable Solution for Similarity Search. In: 2009 Second SISAP. S. 65–73, Aug. 2009.

[SA09] Striemer, G. M.; Akoglu, A.: Sequence alignment with GPU: Performance and design challenges. In: 2009 IEEE SPDP. S. 1–10, Mai 2009.

[Sa10] Sart, D.; Mueen, A.; Najjar, W.; Keogh, E.; Niennattrakul, V.: Accelerating Dynamic Time Warping Subsequence Search with GPUs and FPGAs. In: 2010 IEEE ICDM. S. 1001–1006, Dez. 2010.

[SC90] Sakoe, H.; Chiba, S.: Readings in Speech Recognition. In (Waibel, A.; Lee, K.-F., Hrsg.): Readings in Speech Recognition. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Kap. Dynamic Programming Algorithm Optimization for Spoken Word Recognition, S. 159–165, 1990, ISBN: 1-55860-124-4.

[TDB10] Tomov, S.; Dongarra, J.; Baboulin, M.: Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. Parallel Comput. 36/5-6, S. 232–240, Juni 2010, ISSN: 0167-8191.

[YZG13] You, S.; Zhang, J.; Gruenwald, L.: Parallel Spatial Query Processing on GPUs Using R-trees. In: Proceedings of the 2Nd ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data. BigSpatial '13, ACM, Orlando, Florida, S. 23–31, 2013, ISBN: 978-1-4503-2534-9.

B. Juurlink, W.Karl (Hrsg.)
Proceedings 28th PARS Workshop

Generating Optimized FPGA Based MPSoCs to Parallelize Legacy Embedded Software with Customizable Throughput

Kris Heid[1], Christian Hochberger[1]

Abstract: Executing legacy software on newly developed systems can lead to problems regarding the required throughput of the software. Automatic software parallelization can help to achieve a desired exection time even if a single core version would be to slow. In this contribution, we present a toolset that automatically parallelizes a given legacy software and distributes it to multiple soft-cores forming a processing pipeline. As a goal for the parallelization, the user can provide a minimum throughput that has to be achieved. Although this concept is limited to repetitive tasks, it can be well applied to most embedded system applications. The results show that the tool achieves remarkable speedups without any manual intervention or code restructuring for a sprectrum of benchmarks.

Keywords: automatic parallelization; embedded; soft-core; MPSoC; µStreams

1 Introduction

Reusing existing software is often proposed as a good way to reduce the cost of newly developed embedded systems. Such existing code might implement an application that must obey (soft) real time conditions. To meet these conditions, the performance of a single processing core might not be sufficient and distributing the code over multiple cores could present a solution to this problem.

Automatic parallelization of software is a research topic for many years now. Yet, most of the resulting tools cannot be used for embedded systems, as they either demand a particular infrastructure or programming interface (like CUDA or MPI) or require special OS support.

In this work, we focus on a particular software structure that in turn can be parallelized fully automatically. We expect that the software executes a repetitive task that can be broken down into smaller execution units (thus forming kind of a processing pipeline). Such a structure is often found in embedded software.

To fully exploit this structure, the number of pipeline stages and the communication between different stages must be adapted to the needs of individual applications. Thus, the shown

[1] Technische Universität Darmstadt, Fachgebiet Rechnersysteme, Merckstr. 25, 64283 Darmstadt, Germany
heid@rs-tu-darmstadt.de, hochberger@rs.tu-darmstadt.de

approach is best suited for application specific System-On-Chip (SoC) solutions. These SoCs can either be implemented as a custom chip, which would demand very high quantities, or it could be implemented on a Field Programmable Gate Array (FPGA) which can easily be used even for small quantities. Leveraging FPGAs also allows easy future extensibility with more peripherals or cores.

On an FPGA, the processing cores should be realized as so called *soft-cores*, making it possible to select an arbitrary number of cores and a custom communication infrastructure depending on the application requirements.

In this contribution, we present our fully automatic tool flow that enables the user to generate a SoC architecture with multiple cores and distribute the legacy software to these cores, such that a user specified throughput is achieved.

The remainder of this paper is structured as follows: The next section discusses the related work, particularly other approaches to parallelize software. Section 3 explains our used target architectures. Section 4 then presents our tools and how they work together. An evaluation using multiple benchmarks from different domains is given in Section 5. Finally, a conclusion and an outlook follow.

2 Related Work

Parallelization approaches have already been studied for the past 30 years. For parallelization, different concepts have been proposed: Domain specific languages (DSLs), language extensions, application programming interfaces (APIs), libraries, annotation based parallelization and automatic parallelization. In the order of appearance, the concepts usually make the initial effort for parallelization smaller and smaller, since it is hard to learn new programming languages or explore possibilities of a new API. Nowadays, annotation based parallelizers are relatively popular due to small user effort and good results. OpenMP[CJvdP07] is the de facto standard in this domain and it is adapted to many platforms. This is mainly since OpenMP managed to integrate step by step many features proposed by former competitors like StarSs[La12] with heterogeneous platforms or the system presented by Yang et al.[YL09] with the focus on distributed-memory architectures.

However, automatic parallelization is still the holy grail since it requires no user effort. Many automatic compilers like SUIF[Ha96], Intel C Compiler, Pluto[Bo08] or Daedalus[Th07] only focus on parallelizing (affine nested) loops, which only meets the characteristics for some applications. Other tools like PIPS[KJA15], Eldorado[CMM10] and AutoPar[Li10] extend their parallelization capabilities to the whole program. AutoPar not only focuses on loop parallelization, but also on creating tasks from functions through annotating the source-code with OpenMP loop and task pragmas. PIPS creates a data dependency graph from the source-code. The runtime of the source-code parts is estimated through assembler code analysis and parts of the source-code are mapped to the hardware with a resource aware

list scheduling. PIPS extends the source-code with OpenMP pragmas or generates message passing interface (MPI) code. Eldorado generates a hierarchical control flow graph (CFG) and applies integer linear programming to find an optimal solution with respect to available processor threads, task creation overhead and communication overhead. Compared to PIPS, Eldorado uses a, presumable more inaccurate, average execution time per C source-code statement.

All aforementioned parallelization tools have the following flaws that we want to address in this work:

- Most tools try to extract as much parallelism as possible. Specifying a desired speedup and creating appropriate parallelizations in terms of hard- and software is not considered.
- Data and task parallelism is mainly used. Pipeline parallelism is rarely used even though it is very beneficial for regularly repetitive tasks (like in many embedded systems).
- Even parallelizers targeting embedded systems do not focus on conflicts through simultaneous peripheral access by multiple cores.

3 Target Platforms

As target platform we have used two soft-core multi-processor system-on-chips (MP-SoCs) kits:

SpartanMC is our own 18-Bit architecture where we have full control over the environment. The uncommon 18-Bits optimally leverage todays FPGAs, whose internal structures ideally fit 18 bits. The SoC kit contains a system-builder, simulator, GCC, Binutils, GDB, and synthesis toolchain support for Xilinx, Altera(Intel) and Lattice FPGAs.

MicroBlaze is Xilinx' 32-Bit soft-core. In contrast to other vendor soft-cores like Altera's Nios and LatticeMicro32, it is widely used and in our experience the support for MP-SoCs is better. The available tool environment has similar extend as SpartanMC.

A MP-SoC in both architectures is a distributed-memory system with communication through custom point-to-point as well as point-to-multipoint communication peripherals. Xilinx provides the Mailbox for bidirectional FIFO based communication. The Mailbox can either be configured with an AXI4-Lite interface, implementing memory-mapped IO or an AXI-Stream interface, requiring special processor instructions to access it. The Stream interface should be used for throughput demanding applications, since it has a significantly higher throughput and lower latency. For point-to-multipoint connections we use multiple Mailboxes in parallel.

SpartanMC has a 1-1 core-connector, 1-N dispatcher, N-1 concentrator and an optional shared memory region as communication peripherals. The core-connector is very similar to the Mailbox, except being unidirectional. The concentrator and dispatcher have a round-robin arbiter or software based arbitration to distribute or receive data in a fair manner. All SpartanMC interconnect peripherals are either memory-mapped FIFO buffers or implemented as DMA peripheral memory regions. DMA core-interconnects in SpartanMC require at least two dual-ported RAM primitives (BRAMs), one output of each RAM is connected to core 1, the other to core 2, integrating seamlessly into the core's address space. Only one RAM port to each core is active at a time and a data transfer simply changes the active port on both sides. Thus, the duration of a data transfer of arbitrary size takes 1 cycle, while the non DMA variant requires about 2 cycles per 18bit word. However, BRAM is often a very limited resource and the RAM size must be chosen big enough to contain the maximum message size, while message size is irrelevant for the FIFO based transmission.

4 Tool Flow

We proceeded in different steps with a dedicated tool for each step to realize a parallelized hard- and software design. This decision makes the approach easily adaptable to other Soft-Core MP-SoCs. Additionally, human readable and modifiable files are generated after each intermediate step. This allows the user to step in and modify the design to his wishes or requirements. The overall approach is shown in Fig. 1 and explained in the following.

4.1 AutoPerf: Application Profiling

The legacy sequential source-code is firstly profiled with AutoPerf[HWH18a]. The code is instrumented with calls to the SoCs performance-counter or timer. Each statement is embraced by a call to start the performance-counter and afterwards read and reset it. By default, the main function is profiled, but the user can decide to profile other functions through pragma annotations as-well. The instrumented source-code and an abstract hardware configuration is provided as output. The single-core design can be synthesized and run on the FPGA after automatically importing and building a design in the system-builder (jConfig). The user has to take care to provide an average or worst case environment for the peripheral interaction during measurement. A performance-profile is printed via UART or similar after the application finished.

4.2 AutoStreams: Automatic Annotations

The produced performance profile and the original source-code is feed to AutoStreams[HH18]. The user then specifies the desired throughput of the application. AutoStreams will parse the source-code into an Abstract Syntax Tree (AST). A CFG of the

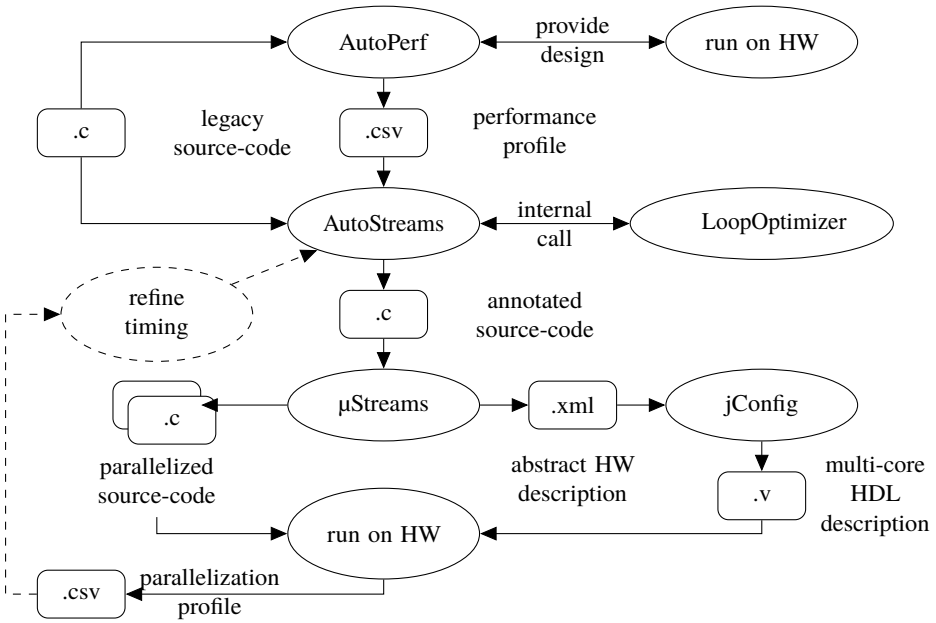


Fig. 1: Automatic parallelization toolflow

profiled source-code parts is created. A CFG node then represents a statement of the profiled function. Often, loops consume vast parts of the overall application runtime. Loops are automatically partitioned into multiple smaller loops with the LoopOptimizer (see Section 4.2.1) if required. This decreases the time granularity and increases amount of CFG nodes.

Afterwards, AutoStreams tries to partition the application to use as few hardware as possible to achieve the user defined throughput. To prefer the smallest possible hardware configuration, an analytical model for each core and core-interconnect has been previously evaluated and added to a AutoStreams hardware usage table. Additionally, the required communication time per communicated data size has been evaluated and is reflected in an approximation function for each core-interconnect. Through these analytical models, an approximation can be done whether it is better to use more cores with slow communication or less cores with fast, but hardware (BRAM) costlier communication peripherals for example.

Now, design space exploration can start. Evaluation of all possible solution takes to long, due to the vast amount of possible solutions. Thus, a branch-and-bound method was chosen. Firstly, a non optimal search heuristic is applied, delivering a solution limiting the search space in the second phase for an optimal solution (detailed description in [HH18]).

To select the best of all solutions fulfilling the timing requirements, firstly pipelines with unique peripherals per pipeline stage are selected, since access to the same peripheral

from different cores is not safely possible [HWH16b]. This behavior can be ignored via command line flags. Secondly, configurations with smallest hardware (LUT) overhead are preferred. Thirdly, amongst minimal hardware designs, the ones with the highest throughput are chosen. Thus, the most economic solution fulfilling user requirements is selected.

The used partitioning is reflected in injected source-code annotations which can be reviewed and if desired manipulated by the user.

4.2.1 LoopOptimizer: Loop Optimizations

Creating balanced pipelines doesn't work well when single statements or CFG nodes (often loops) dominate execution time. Placing pragmas inside loops will create pipeline backward dependencies, prohibiting the benefits of the pipeline. The LoopOptimizer[HWH18b] implements loop splitting and loop fission to provide AutoStreams with loop restructuring and more possibilities of split points.

Loop fission finds independent statements in a loop. All independent statements are partitioned into a separate loop while the loop condition remains identical for all loops.

Loop splitting is a method to distribute the iterations of the original loop to several loops handling a fraction of the original iterations. Thus, the iterations are partitioned while the body stays the same.

4.3 µStreams: Annotated Source-Code Transformation

The target of µStreams[HWH16a] is to split up the original source-code at the pragma annotations into several chunks, forming a processing pipeline. Each pipeline stage will do a fraction of the work of the original application and pass results to the next pipeline stage proceeding in the same way. Thus, the first core is able to handle new data inputs in shorter intervals, increasing the applications throughput. Currently, µStreams has only one pragma: `#pragma microstreams task` with the option to specify `replicate` **number of replicas** to make non dividable pipeline stages superscalar. Pragmas can be placed before any statement, function and inside called functions, but not inside loops or recursive functions. Dependencies between the partitioned source-code chunks are automatically identified and communication infrastructure in software and hardware is automatically created. µStreams is also able to detect used peripherals at the different source code parts based on used APIs and variable types [HWH16b]. One firmware file per core is written as C source-code at the end of modification. Additionally, an abstract hardware description (XML) is created, specifying processor cores, core-interconnects and peripherals. The user also has the option to add evaluation peripherals, which automatically measure the performance of the parallelized design. The abstract hardware description and the firmware sources can be imported into the system-builder (jConfig), automatically connecting and

building the components. The system can then be synthesized and compiled to be run on an FPGA.

4.4 Refine Timing Constraints

The user can review the optionally, but automatically generated parallel performance profile after execution on the FPGA and match it against the requirements. We identified two causes for not fulfilled throughput requirements: Different GCC compiler optimizations and different complex loop iterations of partitioned loops. Varying GCC compiler optimizations can be applied when compiling the parallelized and partitioned code separately, compared to the profiled initial single core compilation, resulting in slower or faster program execution. Partitioned loops in AutoStreams are treated as if each iteration has the same runtime share of the overall loop runtime. This is of course not always the case but a valid approximation for most loops. In both cases, restricting throughput requirements will create a valid design after a few refinement iterations.

4.5 Contributions of this Work

Compared to previous publications of the presented tool-chain, we have made advancements in the following sections:

- We have extended all tools from SpartanMC support to Xilinx MicroBlaze, another soft-core MPSoC. This should show the wider applicability of the approach.
- For the SpartanMC, DMA like core-interconnect peripherals have been created to further decrease critical inter-core communication overhead. The peripherals have been integrated in Verilog and the peripheral characteristics have been added in the analytical model of AutoStreams.
- The pipeline replication mechanism has been integrated in µStreams and AutoStreams to enable superscalar pipeline stages. For replicated pipeline stages the 1-to-N and N-to-1 peripherals are used to distribute and collect application state. We want to analyze the benefits and drawbacks of these constructs and how well the superscalar pipeline improves throughput compared to the previously used pipeline.

5 Evaluation

5.1 Used Benchmarks

To determine the usefulness of the proposed toolchain, we evaluate four benchmark programs: ADPCM, MJPEG2000, IIR filter and a firewall.

Adaptive differential pulse-code modulation (ADPCM) is a digital signal compression algorithm widely used in mobile low-power wireless sensing applications. Infinite impulse response (IIR) Butterworth Filter represents a 5th order high-pass filter. IIR filters are typically used instead of FIR filters for filtering time continuous signals in embedded environments since they require significantly less processing power at an equivalent accuracy. Motion JPEG 2000 Encoder (MJPEG2000) is a video compression algorithm. Compared to other video compression algorithms MJPEG2000 has no inter frame dependencies making it well suitable for parallelization. The firewall implementation contains static as well as dynamic filter rules (managing open TCP connections). We used our research group's (~20 people) firewall as sample for the number of static rules and open TCP connections, since the performance of a firewall mainly depends on the number of rules.

5.2 Results

Tab. 1: Results of automatic parallelization and according hardware cost

		SpartanMC				MicroBlaze		
Benchmark	req.[1]	act.[2]	LUTs	DMA	Cores	act.[2]	LUTs	Cores
ADPCM	2	2.2	3.1	x	3	2.7	3.0	3
	4	3.4	5.2	x	5	4.4	5.0	5
	8	7.2	10.3	x	10	8.0	9.1	10
	12	10.8	15.6	-	17	15.7	20.3	20
IIR Filter	2	2.1	2.7	-	3	2.0	2.3	3
	4	3.9	5.2	x	5	3.8	9.3	12
	8	6.0	12.5	x	12	exceeds FPGA res.		
	12	8.1	25.2	x	24	exceeds FPGA res.		
MJPEG	2	2.6	3.0	-	3	2.2	3.0	3
	4	4.4	6.3	-	6	3.5	5.0	5
	8	exceeds FPGA resources						
Firewall	2	3.1	2.6	-	4	1.8	2.3	4
	4	4.5	3.0	-	5	3.5	3.2	6
	8	7.4	3.5	-	7	5.1	5.1	10
	12	7.4	3.5	-	7	5.2	6.0	12

[1] user requested speedup | [2] actual speedup after parallelization

For the benchmarks, we set a user performance requirement to AutoStreams that should be achieved. The timing requirement reflects a 2x, 4x and if possible 8x and 12x higher throughput compared to the single core performance throughput of SpartanMC or MicroBlaze respectively. We allowed AutoStreams to use DMA and non DMA core-interconnects and superscalar core replication. After parallelization, we've synthesized the generated design for a Xilinx Artix-7 XC7A200T FPGA and finally ran the design while evaluating hardware overhead and achieved speedup.

As seen in Tab. 1, the performance demands of the 2x speedup requirements has almost always been fulfilled. To achieve a 2x speedup, at least three processing cores are leveraged. In the ideal case, two cores should be sufficient to achieve this speedup if we neglect communication overhead. Thus, a quite high overprovisioning was chosen. As speedup requirements increase, a smaller overprovisioning is chosen, since AutoStreams takes the smallest possible hardware just fulfilling the throughput requirements. However, due to the applied single core compiler optimizations which might not be applicable anymore on the partitioned parallelized code, AutoStreams thinks the requirements are met while the measured throughput is below the requirement. Surprisingly, the SpartanMC observed throughput was often 20% to 30% below the the expected throughput, while for MicroBlaze systems the observed throughput was only around 10% below the expected. Most likely the cause lies in the different compiler optimizations for both systems, since deactivating compiler optimizations on one benchmark revealed no discrepancy between the estimated and the actual throughput. Thus, for all benchmarks the user could make a second parallelization run with accordingly tightened timing restrictions.

Considering the hardware cost of parallelized design, one can see that for a higher parallelization unproportionally more cores are required. Firstly, it is often impossible to generate a balanced pipeline where each core has the same workload due to the finite granularity of the working packages to distribute. Secondly, the application state has to be passed through the pipeline which is the communication overhead. The longer the processing pipeline gets, the smaller are the workloads per core and thus communication overhead grows relatively. Comparing the IIR benchmark (much application state information) and the ADPCM benchmark (few application state information) one can see that for a 8x and 12x speedup, IIR needs a longer processing pipeline to fulfill the requirements. However, looking at the increase of actual used hardware (LUTs) on the FPGA, one can see that this number does not always grow proportionally with the amount of cores. For example the SpartanMC 17-core ADPCM example only uses 15x more LUTs compared to the single core variant, since the synthesis is able to reuse some hardware. In this evaluation we've just focused on the used LUTs but the other FPGA components (BRAMs, registers and DSPs) grow at similar rates.

We have restricted the available FPGA's resources for the ADPCM benchmark's SpartanMC builds to 200 BRAMs (since BRAMs are often the limiting resource), simulating a smaller FPGA and intentionally triggering AutoStreams to use BRAM preserving non DMA core-interconnects on resource shortage. This resulted in the 12x speedup case to use non DMA core-interconnects and thus still be synthesizable on a smaller FPGA.

The IIR example reveals a different aspect. The 2x speedup requirement is realized though non DMA core-interconnects for the SpartanMC. However, for tighter user requirements (4x,8x,12x), DMA variants are chosen. Compared to the workload size, the communication overhead grows, favoring the faster DMA core-interconnects. To achieve the same speedup with non DMA core interconnects an additional processing core would be required costing more hardware than the DMA-core interconnects. Also, a 12x speedup parallelization

without DMA interconnects is not possible since the communication overhead is bigger than the pipeline stages duration requirement. For the MicroBlaze variant, even a 8x speedup is not possible anymore since this would result in a 50 core design vastly busting the bounds of our used FPGA.

The MJPEG2000 benchmark requires a lot of hardware resources (BRAMs) since the full frame is transferred through the pipeline and stored in each core. Also, the communication overhead is quite big through the transferal of the whole frames. DMA core-interconnects for the SpartanMC system can not be used since the available address space would be exceeded. Through the large communication overhead, the necessary core number quickly increases and through the required memory per core, the FPGA's available BRAM is exceeded, preventing further parallelization. Even though not shown, AutoStreams suggests the usage of replicas/superscalar pipeline stages for a 8x speedup since inseparable code parts would exceed the timing requirements. Comparing the hardware cost of a regular pipeline with a superscalar pipeline revealed that a regular pipeline uses fewer hardware at the same throughput. This fact is also reflected in AutoStreams observed behavior to only use replication if no further sequential pipeline stages are possible.

The firewall benchmark's generated hardware consists of a packet receiving core, multiple filter cores connected through a 1-N dispatcher and a packet sending core. Thus, all filter cores form a superscalar pipeline stage through the replicate pragma. The filter rules are stored in a common memory module connected to all cores. The core receiving and the one sending Ethernet packets become the bottleneck, visible through a saturation effect for a 8x and 12x speedup requirement. The sending and receiving cores can't be parallelized since they are directly interfacing the Ethernet hardware modules and AutoStreams even throws a warning that the desired speedup is unreachable. But the saturation effect also comes from the common filter table memory where all filter cores compare the packet header against.

6 Conclusion

In this contribution we have presented a toolset for the fully automatic parallelization of legacy software. The user only has to specify a required throughput and the tool then finds a suitable HW setup of a multi-core system with minimal area requirement and the corresponding distribution of the legacy software. Even if the expected software structure is limited, the tool is applicable to most embedded systems applications.

In the future, we want to improve the communication primitives that are used together with the MicroBlaze core. The Mailbox provided by Xilinx exhibits quite a large latency. Also, we think that the handling of global variables in the legacy code can be improved.

References

[Bo08] Bondhugula, Uday; Baskaran, Muthu; Krishnamoorthy, Sriram; Ramanujam, J.; Rountev, A.; Sadayappan, P.: Automatic Transformations for Communication-Minimized Parallelization and Locality Optimization in the Polyhedral Model. In: ETAPS CC. 2008.

[CJvdP07] Chapman, Barbara; Jost, Gabriele; van der Pas, Ruud: Using OpenMP. Mit University Press Group Ltd, 2007.

[CMM10] Cordes, D.; Marwedel, P.; Mallik, A.: Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In: International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS). 2010.

[Ha96] Hall, M. W.; Anderson, J. M.; Amarasinghe, S. P.; Murphy, B. R.; Liao, Shih-Wei; Bugnion, E.; Lam, M. S.: Maximizing multiprocessor performance with the SUIF compiler. Computer, 29(12):84–89, Dec 1996.

[HH18] Heid, Kris; Hochberger, Christian: AutoStreams: Fully Automatic parallelization of Legacy Embedded Applications with Soft-Core MPSoCs. In: ReConFig 2018; International Conference on Reconfigurable Computing and FPGAs. Dec 2018.

[HWH16a] Heid, Kris; Weber, Jan; Hochberger, Christian: µStreams: A Tool for Automated Streaming Pipeline Generation on Soft-core Processors. In: 2016 International Conference on FPGA Reconfiguration for General-Purpose Computing (FPGA4GPC). pp. 25–30, May 2016.

[HWH16b] Heid, Kris; Wirsch, Ramon; Hochberger, Christian: Automated Inference of SoC Configuration through Firmware Source Code Analysis. In: FSP 2016; Third International Workshop on FPGAs for Software Programmers. pp. 1–9, Aug 2016.

[HWH18a] Heid, Kris; Wenzel, Jakob; Hochberger, Christian: Fast DSE for Automated Parallelization of Embedded Legacy Applications. In: Applied Reconfigurable Computing. Architectures, Tools, and Applications. pp. 471–484, 2018.

[HWH18b] Heid, Kris; Wenzel, Jakob; Hochberger, Christian: Improved Parallelization of Legacy Embedded Software on Soft-Core MPSoCs through Automatic Loop Transformations. In: FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers. Aug 2018.

[KJA15] Khaldi, Dounia; Jouvelot, Pierre; Ancourt, Corinne: Parallelizing with BDSC, a resource-constrained scheduling algorithm for shared and distributed memory systems. Parallel Computing, 41:66 – 89, 2015.

[La12] Labarta, Jesús; Marjanovic, Vladimir; Ayguadé, Eduard; Badia, Rosa M; Valero, Mateo: Hybrid Parallel Programming with MPI/StarSs. IOS Press, May 2012.

[Li10] Liao, Chunhua; Quinlan, Daniel J.; Willcock, Jeremiah J.; Panas, Thomas: Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions. International Journal of Parallel Programming, 2010.

[Th07] Thompson, M.; Nikolov, H.; Stefanov, T.; Pimentel, A. D.; Erbas, C.; Polstra, S.; Deprettere, E. F.: A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In: CODES+ISSS. 2007.

[YL09] Yang, Chao-Tung; Lai, Kuan-Chou: A Directive-based MPI Code Generator for Linux PC Clusters. J. Supercomput., 50(2):177–207, November 2009.

B. Juurlink, W. Karl (Hrsg.)
Proceedings 28th PARS Workshop

A Quantitative Analysis of Processor Memory Bandwidth of an FPGA-MPSoC

Robert Drehmel[1], Matthias Göbel[2], Ben Juurlink[3]

Abstract: System designers have to choose between a variety of different memories available on modern FPGA-MPSoCs. Our intention is to shed light on the achievable bandwidth when accessing them under diverse circumstances and to hint at their suitability for general-purpose applications. We conducted a systematic quantitative analysis of the memory bandwidth of two processing units using a sophisticated standalone bandwidth measurement tool. The results show a maximum cacheable memory bandwidth of 7.11 GiB/s for reads and 11.78 GiB/s for writes for the general-purpose processing unit, and 2.56 GiB/s for reads and 1.83 GiB/s writes for the special-purpose (real-time) processing unit. In contrast, the achieved non-cacheable read bandwidth lies between 60 MiB/s and 207 MiB/s, with an outlier of 2.67 GiB/s. We conclude that for most applications, relying on DRAM and hardware cache coherency management is the best choice in terms of benefit-cost ratio.

Keywords: Memory Bandwidth; MPSoC; Interconnects

1 Introduction

As MPSoCs evolve into more complex devices containing increasingly heterogeneous processing units, a growing variety of specialized memories becomes available to developers. Although logically connected to a single system bus, the processors and memories in modern MPSoCs are physically connected using an intricate net of interconnects with distinctive performance characteristics. As interconnects are often seen as the limiting factor of SoC performance[1], being aware of their influence on bandwidth can play an important role in the process of designing a system. The achievable bandwidth when accessing different memories depends on various factors, e.g. the processor's performance, the pathway of interconnects between the processor and the memory, and the cacheability associated with the memory region. To make informed decisions about which memory to use for a given task, developers need to understand their capabilities and limitations and need to be able to compare their performance systematically.

[1] Technische Universität Berlin, Embedded Systems Architecture, Einsteinufer 17 (6. OG), 10587 Berlin, Germany
drehmel@campus.tu-berlin.de
[2] Technische Universität Berlin, Embedded Systems Architecture, Einsteinufer 17 (6. OG), 10587 Berlin, Germany
m.goebel@tu-berlin.de
[3] Technische Universität Berlin, Embedded Systems Architecture, Einsteinufer 17 (6. OG), 10587 Berlin, Germany
b.juurlink@tu-berlin.de

The main contribution of this paper is a systematic and comprehensive evaluation of the processor memory bandwidth achievable using the different processing units of the Xilinx Zynq Ultrascale+ MPSoC line.

This paper is organized as follows. Section 2 discusses related work, section 3 gives a brief overview of the platform used, section 4 outlines the design and implementation of the bandwidth measurement tool, section 5 details the experimental setup and presents the results of the evaluation. Finally, section 6 draws conclusions based on the results.

Note that in the following, when using the term *memory bandwidth*, we mean the observed bandwidth (or throughput) when accessing a certain memory, including and emphasizing the multiplicity of components involved in the process, like caches, store buffers, interconnects, memory controllers, and the memory itself.

2 Related Work

Göbel et al.[2] conducted thorough system bus bandwidth analyses of chips from the previous generation of FPGA-MPSoCs: the Intel Cyclone-V, the Xilinx Zynq-7020, and the Zynq-7045. Choi et al.[3] evaluated specialized systems with an FPGA and processor combination, i.e. Intel Xeon E5-2680v2/Stratix V on an Intel HARP board and Intel Xeon E5-2620v3/Xilinx Virtex 7 on an Alpha Data board.

Closely related to our work, Bansal et al.[4] recently evaluated the memory subsystem of the Zynq Ultrascale+, but with a focus on giving advice for the design of real-time applications. They measured bandwidth only for a single processing unit running Linux to three different memories for a specific amount of time (5 seconds) and using unspecified instructions generated by the compiler. In contrast, we measured sequential accesses for varying transfer sizes to four different memories using a sophisticated standalone application, for two processing units, and for two types of instructions.

3 Xilinx Zynq Ultrascale+

A chip of the Zynq Ultrascale+ series provides an Application Processing Unit (APU) and a Real-Time Processing Unit (RPU)[5]. The APU consists of an ARM Cortex-A53 processor[6] with four cores, each with 64 KiB L1 cache (separate 32 KiB for instructions and data) and a shared 1 MiB L2 cache. The RPU consists of two single-core ARM Cortex-R5 processors[7], each with 32 KiB L1 cache (combined for instruction and data). One of the two main regions of the chip, the *Processing System* (PS), contains common MPSoC components; the other one, the *Programmable Logic* (PL), holds the integrated FPGA fabric. We used the Zynq Ultrascale+ ZU9EG chip for our tests. It supplies the system designer with memories of four main categories, namely

- external Dynamic RAM (DRAM) through an memory controller integrated in the PS,

- 256 KiB On-Chip Memory (OCM),
- 256 KiB Tightly-Coupled Memory (TCM), 128 KiB coupled to each Cortex-R5 of the RPU, again divided into two separate 64 KiB blocks (ATCM and BTCM),
- 32.1 MiB Block RAM (BRAM) in 912 blocks in the PL.

Some chips of the series include so-called UltraRAM on-chip memory in the PL. Unfortunately, this is not the case with the ZU9EG, therefore we were not able to include this kind of memory in our tests.

4 Benchmark Tool

The core functionality of our benchmark tool is to measure the time it takes the processor to complete the execution of a bandwidth test function in a specific execution context. The context of the execution of the test function – and the arguments passed to it – are highly parameterizable.

The tool allows the user to set parameters to

- select various forms of cacheability for inner and outer domains (ARM-specific),
- select the ISA to use for the memory access (base or Adv. SIMD)[8][9],
- select the access type (read or write),
- select the width of the memory transfer (in powers of two),
- enable or disable the data cache,
- specify the number of rounds of reading and writing before the measurements (to fill the cache with read-allocate and write-allocate, respectively),
- select the shareability domain (ARM-specific), and
- enable or disable APU coherency (Zynq Ultrascale+-specific).

The user can define a stack of functions to automatically gather results for a set of parameters. Each function included in the stack iterates through all possible values of a single parameter and calls the next function in the stack after setting a new value. The last function in the stack is the test run function that evaluates all the currently set parameters, sets up the context (e.g., hardware configuration and memory attributes) accordingly and runs the test. The results for each test run are saved along with the parameters used for the test. After all tests completed, the user can query the result database for further processing, for example to perform advanced analyses to find statistical anomalies or to generate plots in LaTeX documents.

To retain full control of the hardware configuration, we developed our benchmark tool as a bare-metal application using Xilinx's *standalone* library. The tool currently supports Cortex-A53 and Cortex-R5 processors through a hardware abstraction layer that provides functionality like management of hardware cycle counters, caches, Memory Protection Units (MPUs), and Memory Management Units (MMUs).

The hardware abstraction layer also provides a set of hand-optimized read and write benchmark test functions. For a given processor, if $2^{W_{\text{bus}}}$ bytes is the width of the master

Processor	Mnemonic	Access width (bytes)	ISA	Type
APU	`LDP`	$2 \cdot 8 = 16$	ARMv8-A	load
APU	`STP`	$2 \cdot 8 = 16$	ARMv8-A	store
APU	`LD1`	$8 \cdot 8 = 64$	ARMv8-A Adv. SIMD	load
APU	`ST1`	$8 \cdot 8 = 64$	ARMv8-A Adv. SIMD	store
RPU	`LDM`	$8 \cdot 4 = 32$	ARMv7-R	load
RPU	`STM`	$8 \cdot 4 = 32$	ARMv7-R	store
RPU	`VLDM`	$8 \cdot 8 = 64$	ARMv7-R Adv. SIMD	load
RPU	`VSTM`	$8 \cdot 8 = 64$	ARMv7-R Adv. SIMD	store

Tab. 1: Instructions used in the bandwidth test functions

interface to the system bus, and $2^{W_{\text{insn}}}$ bytes is the largest number of bytes transferable with a single instruction, for each n in $[W_{\text{bus}} : W_{\text{insn}}]$ an optimized function is provided that transfers 2^n bytes in one loop iteration. Assuming $W_{\text{bus}} \leq W_{\text{insn}}$, for a test width of W_{test} bytes, the function that is optimized to transfer W_{f} bytes is selected, where

$$W_{\text{f}} = \begin{cases} W_{\text{bus}} & : W_{\text{test}} \leq W_{\text{bus}} \\ W_{\text{test}} & : W_{\text{bus}} < W_{\text{test}} < W_{\text{insn}} \\ W_{\text{insn}} & : W_{\text{test}} \geq W_{\text{insn}} \end{cases}$$

Each function is provided in all four possible combinations for read/write access types and base ISA/Adv. SIMD instructions. Table 1 shows the instructions used in the bandwidth test functions and their corresponding access widths.

To configure the caching behavior for the Cortex-A53, the tool sets the memory attributes in the page table entries corresponding to the tested memory region to *normal memory, inner/outer non-cacheable* and *normal memory, inner/outer write-back* to test non-cacheable and cacheable accesses, respectively. We found that disabling the data cache (by clearing the `C` bit in the `SCTRLR_ELx` register) can have a different effect (i.e. reduced bandwidth) than marking a region non-cacheable in its page table entries. We attribute this to the fact that clearing the abovementioned bit disables the data cache and the unified caches, and has non-intuitive effects such as preventing the caching of page table memory.

The boot code of the standalone library for the Cortex-R5 installs a default MPU configuration that includes a region that spans the first 2 GiB of the address space. On the Zynq Ultrascale+, the ATCM is mapped to `0x0` for each Cortex-R5. The code and data segments, the heap, and the stack of the bandwidth tool are held in a region starting at `0x10000000`. We use a region starting at `0x20000000` to test the DRAM. So instead of a single memory region that spans the first 2 GiB of the address space, beginning at the start of the address space, the tool configures three consecutive memory regions that each span 256 MiB. This is necessary to change the cacheability attributes for the TCM and the DRAM without interfering with the cacheabilty of the region the bandwidth tool runs in.

5 Evaluation

In the following, we first detail the experimental setup and subsequently present the quantitative results gathered.

5.1 Experimental setup

The benchmark tests use the following fixed parameters: enabled data cache, four rounds of prefilling the data cache with reads (read-allocate), zero rounds of prefilling the data cache with writes (write-allocate), outer shareability domain, and enabled APU coherency. We are basing our decision to leave the APU coherency enabled on the premise that most system designers will keep it enabled – it is the default setting and is useful for all except a minority of special applications.

We ran tests for every combination of the parameters *processor* (APU/RPU), *caching* (write-back/non-cacheable), *memory* (DRAM/OCM/ATCM/BRAM), *instruction type/ISA* (base ISA/Adv. SIMD), and *access type* (read/write), and *transfer size* ($N + 1$ different transfer sizes, 2^0 to 2^N bytes, where 2^N bytes is the size of the memory region tested). Every test was run 1000 times and averaged accordingly.

We tested memory accesses to 16 MiB DRAM, 256 KiB OCM, 64 KiB ATCM, and 1 MiB BRAM. The ZCU102 board used for the evaluation has 4 GiB of DDR4 memory (2133 MHz) installed. Although the four 64 KiB TCMs available on the chip can be mapped into a consecutive 256 KiB region, this is only possible when both Cortex-R5 processors are running in *lock-step* mode (coupled execution of both processors). A more common mode of operation is the antithetic *split* mode (separate execution of both processors), in which each processor can only access its own (non-consecutively-mapped) ATCM and BTCM. These and other particularities of the TCM as implemented in the Zynq Ultrascale+ are described in detail in [5]. We decided to choose a single 64 KiB ATCM as the TCM test region to cover a broader range of potential applications. The 1 MiB of BRAM (128 bit data width and 64 KiB depth) consist of 256 RAMB36 BRAM blocks (a utilization of roughly 28%) generated by a Xilinx *Block Memory Generator* [10], connected to a Xilinx *AXI BRAM Controller* [11] that is in turn connected to the PS using the HPM0 port.

5.2 Results

Non-cacheable APU memory access. First, we discuss the bandwidth results for non-cacheable APU memory accesses as displayed in figure 1. Results for reads and writes both show the maximum sustainable bandwidth. Non-cacheable read bandwidth only increases with the size of memory transferred (a slope similar to figure 5b). This is a trivial consequence of a decreasing percentage of time spent processing the prologue and epilogue instructions of the test function. Reading using the base ISA provides roughly twice as much

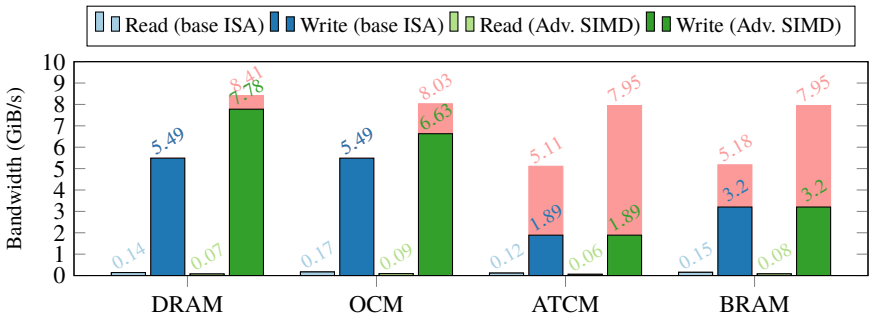
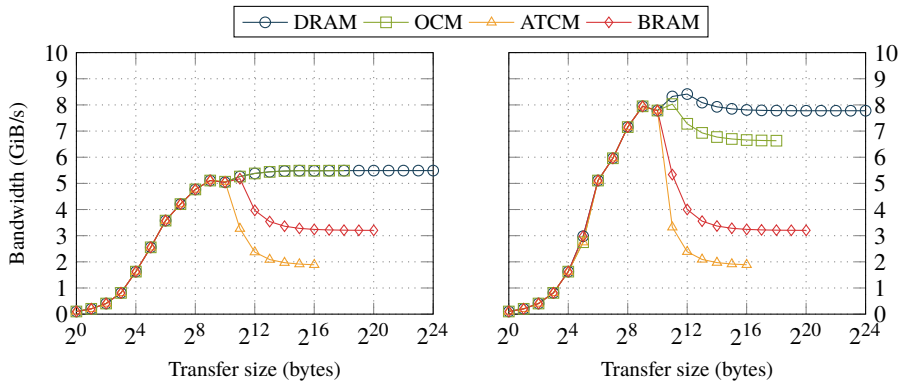


Fig. 1: Bandwidths of non-cacheable APU memory accesses (for the largest tested transfer size). denotes unsustainable or short-term write bandwidth.



(a) Base ISA (b) Adv. SIMD

Fig. 2: Bandwidths of non-cacheable APU write memory accesses

bandwidth as using the Advanced SIMD instructions. Still, none of the non-cacheable APU read bandwidths exceeds 172 MiB/s. Results for write accesses also show the maximum short-term bandwidths. While read instructions need to move data back to the processor core and therefore need to wait for the read request to finish, write requests are simply handed off to another stage. This stage then processes the queued write requests autonomously. If that stage does not have the capacity to store more outstanding write requests, the hand-off from the processor is stalled. This might happen because one or more components in the path to the memory are too slow to keep up or congestion occurs. These throttling effects are visible in figure 2. Using the base ISA write instructions (figure 2a), the throttling effect appears at transfer sizes of 2^{11} bytes and 2^{12} bytes for writes to ATCM and BRAM, respectively, while for DRAM and OCM the limiting factor is the processor core. On the other hand, the Adv. SIMD write instructions (figure 2b) show a throttling effect for all memories: starting

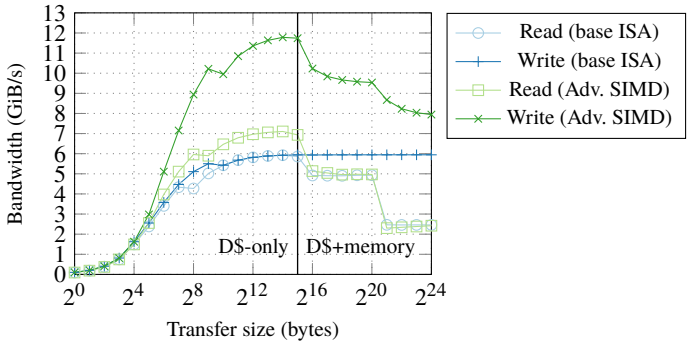


Fig. 3: Bandwidths of cacheable APU memory accesses to DRAM

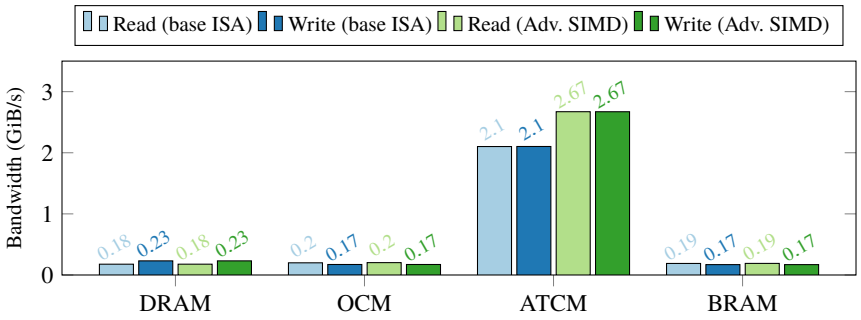
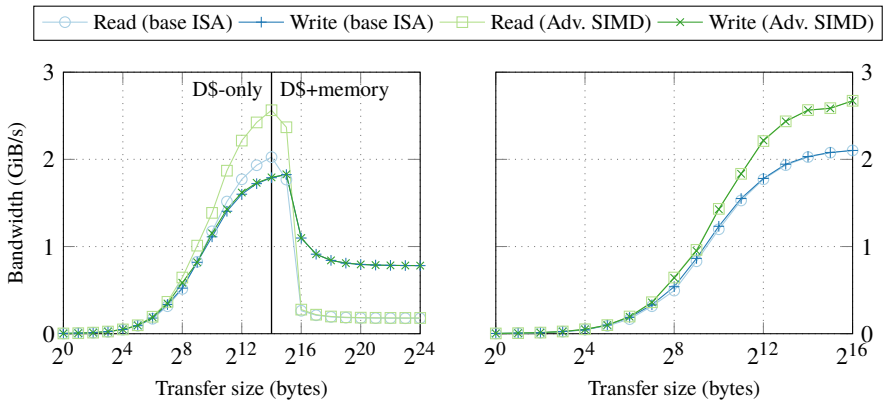


Fig. 4: Bandwidths of non-cacheable RPU memory accesses (for the largest tested transfer size)

at transfer sizes of 2^{11} bytes for ATCM and BRAM, 2^{12} bytes for OCM and 2^{13} bytes for DRAM.

Cacheable APU memory access. Figure 3 shows the cacheable APU memory bandwidths for the DRAM. As the bandwidths of cacheable APU memory accesses up to the transfer size of 2^{15} bytes are practically the same for all memories due to little to no cache misses, we show only the DRAM plot. There is a first significant drop at a transfer size of 2^{16} bytes, where the L1 data cache with its size of 32 KiB can only hold part of the data. Another drop is found at 2^{21} bytes, where the L2 cache size of 1 MiB is exhausted. The read bandwidth peaks at 7.11 GiB/s and the write bandwidth peaks at 11.78 GiB/s, both using Adv. SIMD instructions. It is notable that the Advanced SIMD instructions for writing outperform the base ISA instructions almost by a factor of two at transfer sizes around 2^{14} bytes.

Non-cacheable RPU memory access. Figure 4 shows the maximum bandwidths achievable with non-cacheable RPU memory accesses. The throttling effects exposed by the results



(a) DRAM (b) ATCM

Fig. 5: Bandwidths of cacheable RPU memory accesses

for the non-cacheable APU write memory accesses do not appear with non-cacheable RPU write memory accesses. The reason is not because the Cortex-R5 is not fast enough to force congestion. In fact, it congests an internal component; when the processor encounters a write instruction destined to access a non-cacheable memory region, it submits a write request to its *store buffer* which directly generates an AXI write request [12] on the AXI master interface [7]. The processor then has to wait for the AXI request to complete before it can submit the next write request to the store buffer. Hovering between 181 MiB/s (DRAM) and 207 MiB/s (OCM), read bandwidth to all memories except the ATCM is roughly comparable to non-cacheable APU read bandwidth, apart from the non-existence of a performance gap between base ISA and Adv. SIMD instructions that is present in the non-cacheable APU memory access results. The ATCM unveils its strength in this benchmark with read and write bandwidths of 2.1 GiB/s for ISA instructions and 2.67 GiB/s for Adv. SIMD instructions.

Cacheable RPU memory access. Figure 5a shows the bandwidths of cacheable RPU memory accesses with different transfer sizes to DRAM. Like for the APU, up to a certain transfer size, the bandwidth of cacheable memory accesses is determined by the L1 cache. For the RPU, the transfer size where cache misses and ensuing cache line fills start to noticeably impact the bandwidth is 2^{15} bytes: a minor drop occurs from 2^{14} to 2^{15} bytes, because the 32 KiB L1 cache of the Cortex-R5 is a combined instruction and data cache and therefore instructions use up some of the cache capacity. Starting at transfer sizes of 2^{16} bytes, frequent L1 cache misses drag the bandwidth down immensely as there is no L2 cache. Figure 5b shows a dissimilar behavior for the ATCM as the Cortex-R5 processor always treats accesses to its TCMs as non-cacheable accesses. Accordingly, the bandwidths for transfer sizes of 2^{16} bytes shown in figure 5b match those of the ATCM in figure 4. The

peak cacheable read bandwidth is 2.56 GiB/s (96% of the ATCM read bandwidth) and the peak cacheable write bandwidth is 1.83 GiB/s (69% of the ATCM write bandwidth).

6 Conclusions

We used our benchmark tool to systematically measure the memory bandwidth of the different processing units of an FPGA-MPSoC.

Our results show surprisingly underwhelming non-cachable read memory bandwidths, generally ranging from 60 MiB/s to 207 MiB/s, across the board for both processing units and all non-TCM memories. This extends to non-cachable write memory bandwidths on the RPU, ranging from 174 MiB/s to 237 MiB/s for non-TCM memories. Exceptions are the read and write ATCM bandwidths of up to 2.67 GiB/s, but only when accessing the ATCM from the RPU. Write memory bandwidths can, in theory, be much higher – and are in practice for the APU – but depend heavily on processor-internal request queueing. The APU achieves maximum non-cachable short-term write bandwidths ranging from 5.11 GiB/s to 8.41 GiB/s and is able to sustain a maximum bandwidth of 7.78 GiB/s to DRAM using Advanced SIMD instructions. Cacheable memory accesses from the APU show high maximum bandwidths, reaching 7.11 GiB/s for reads (Adv. SIMD) and 11.78 GiB/s for writes (Adv. SIMD). In comparison, the RPU reaches a maximum cacheable memory bandwidth of 2.56 GiB/s (Adv. SIMD) for reads and 1.83 GiB/s (Adv. SIMD) for writes.

We conclude that the use of dedicated on-chip memory, except tightly-coupled memory, has no additional benefits in terms of bandwidth over external DRAM memory. To achieve high memory bandwidth and therefore high computational performance on the Zynq Ultrascale+ platform, leveraging the use of system-wide hardware cache coherency is therefore recommended. As hardware cache coherency management has the potential – depending on the application – to reduce software complexity, we see a dependence solely on DRAM and hardware cache coherency as the best choice for most applications. On the other hand, using our findings on unsustainable non-cacheable write memory bandwidths, an application could optimize parallelism by writing chunks of memory in specific sizes to fill the write request queue and doing other processing while the write requests are being completed. The caches behave as expected, so the usual recommendations naturally apply, such as exploiting data locality by taking knowledge of specific cache implementation parameters like cache line size into account for the implementation of an application. Advanced SIMD instructions provide at least the same – in some cases vastly superior – bandwidth compared to the base ISA instructions – non-cacheable APU reads being the exception.

To gain a better understanding of interconnect behavior, further work could include methodical stress testing that focusses on inducing interconnect congestion using multiple bus masters, i.e. multiple processor cores or multiple processing units that simultaneous issue memory access requests. Evaluation of UltraRAM memory could also prove to produce interesting results.

Acknowledgements

This research was partially funded by the *German Academic Scholarship Foundation (Studienstiftung des deutschen Volkes)*.

References

[1] Luca Benini and Giovanni De Micheli. "Networks on chips: A new SoC paradigm". In: *Computer* 35 (1 Jan. 2002), pp. 70–78. DOI: 10.1109/2.976921.

[2] Matthias Göbel et al. "A Quantitative Analysis of the Memory Architecture of FPGA-SoCs". In: *Applied Reconfigurable Computing. 13th International Symposium, ARC 2017*. Lecture Notes in Computer Science 10216. Springer, 2017, pp. 241–252. DOI: 10.1007/978-3-319-56258-2_21.

[3] Young-kyu Choi et al. "A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms". In: *2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 2016. DOI: 10.1145/2897937.2897972.

[4] Ayoosh Bansal et al. "Evaluating the Memory Subsystem of a Configurable Heterogeneous MPSoC". In: *Proceedings of OSPERT 2018. the 14th Annual Workshop on Operating Systems Platforms for Embedded Real-Time Applications*. 2018, pp. 55–60.

[5] *Zynq UltraScale+ Device. Technical Reference Manual*. Version v1.9. UG1085. Xilinx. Jan. 17, 2019.

[6] *ARM Cortex-A53 MPCore Processor. Technical Reference Manual*. Version Issue J (r0p4). ARM DDI 0500J (ID071918). ARM. June 13, 2018.

[7] *Cortex-R5. Technical Reference Manual*. Version Issue D (r1p2). ARM DDI 0460D (ID092411). ARM. Sept. 15, 2011.

[8] *ARM Architecture Reference Manual. ARMv7-A and ARMv7-R edition*. Version Issue C.c. ARM DDI 0406C.c (ID051414). ARM. May 20, 2014.

[9] *ARM Architecture Reference Manual. ARMv8, for ARMv8-A architecture profile*. Version Issue D.a. ARM DDI 048D.a (ID103018). ARM. Oct. 31, 2018.

[10] *Block Memory Generator v8.4. LogiCORE IP Product Guide*. PG058. Xilinx. Oct. 4, 2017.

[11] *AXI Block RAM (BRAM) Controller v4.1. LogiCORE IP Product Guide*. PG078. Xilinx. Dec. 5, 2018.

[12] *AMBA AXI and ACE Protocol Specification. AXI3, AXI4, AXI5, ACE and ACE5*. Version Issue F.b. ARM IHI 0022F.b (ID122117). ARM. Dec. 21, 2017.

Influence of Discretization of Frequencies and Processor Allocation on Static Scheduling of Parallelizable Tasks with Deadlines

Sebastian Litzinger, Jörg Keller[1]

Abstract: Models for energy-efficient static scheduling of parallelizable tasks with deadlines on frequency-scalable parallel machines comprise moldable vs. malleable tasks and continuous vs. discrete frequency levels. We investigate the tradeoff between scheduling time and energy efficiency when going from continuous to discrete processor allocation and frequency levels. To this end, we present a tool to convert a schedule computed for malleable tasks on machines with continuous frequency scaling (P. Sanders, J. Speck, Euro-Par 2012) into one for moldable tasks on a machine with discrete frequency levels. We compare the energy efficiency of the converted schedule to the energy consumed by a schedule produced by the integrated crown scheduler (N. Melot et al., ACM TACO 2015) for moldable tasks and a machine with discrete frequency levels. Our experiments indicate that the converted Sanders Speck schedules, while computed faster, consume more energy on average than crown schedules. Surprisingly, it is not the step from malleable to moldable tasks that is responsible, but the step from continuous to discrete frequency levels.

Keywords: Static Scheduling; Frequency Scaling; Energy-efficient Schedules

1 Introduction

Repetitive tasks with deadlines often occur in embedded multicore systems, for example in streaming applications where each task is activated in a scheduling round [Me15]. The length of the round determines the throughput, such as the number of images that can be fed to an image processing application per second, and at the same time poses a deadline until which all the tasks must or should be executed, depending on the hardness of the deadline. While each task feeds output to its follow-up tasks according to the streaming task graph, this input can be considered to be transferred from one scheduling round to the next, so that the task invocations in one round are independent of each other.

If the application has a longer runtime or is frequently executed, it pays off to find a static schedule to execute the tasks within a round with minimum frequencies in order to lower the energy consumption for the given throughput. A static schedule typically has lower runtime overhead and at the same time better quality than a dynamic schedule, given that

[1] FernUniversität in Hagen, Faculty of Mathematics and Computer Science, 58084 Hagen, Germany first.last@fernuni-hagen.de

the task executions are predictable, which is the case in the above scenario. Minimizing energy consumption is of tremendous importance in embedded systems, as lower energy might mean a housing without a fan and thus lower production and maintenance cost. At least it means a lower bill from reduced energy supply and reduced cooling.

An interesting sub-case of this static scheduling scenario are parallelizable tasks, i. e. tasks that are parallel programs themselves. This is advantageous e. g. when the number of cores is larger than the number of tasks. Several approaches have proposed solutions to this problem, however starting from different assumptions. On the one hand, Sanders and Speck [SS12] have proposed an algorithm that computes a schedule under the assumption that the frequency can assume an arbitrary value, and not only a finite number of discrete levels, and under the assumption that the degree of parallelism of a task can vary during its execution (so-called malleable tasks), i. e. a task allocated to 5.3 cores runs on 6 cores for 30% of the time till the deadline, and on 5 cores for the remaining time. Finally, they assume that a (sequential) task can be interrupted and continued later on. On the other hand, crown scheduling [Me15] assumes that only a finite number of discrete frequency levels is available, that a task can only be allocated to an integral number of cores[2] (so-called moldable tasks) and that a task is not interrupted by another task once its execution has started. There are other differences as well (the Sanders Speck scheduler poses some restrictions on the power and speedup functions, while the crown scheduler allows arbitrary power and speedup profiles) which we will ignore here.

In the current work, we are interested in the trade-off between these two extremes. In particular, we investigate the hypothesis that to schedule a set of moldable tasks, we might first use the Sanders Speck scheduler as if the tasks were malleable, and convert the schedule into one for moldable tasks, remove preemption of sequential tasks, and step over to discrete frequencies. For all these steps, we observe how much the energy consumption of the resulting schedule increases, and compare the final result with the energy consumption of a crown schedule. The Sanders Speck scheduler (even including the converter) has a shorter runtime than crown scheduling, which solves a mixed-integer linear program. Our hypothesis, which we will test by experiments, is that a converted Sanders Speck schedule has a higher energy consumption than a crown schedule, so the user can trade scheduling time for energy consumption. In addition, by doing a sequence of conversion steps we can see which of the model differences has most influence, so that one may perform research on that difference in the future.

Surprisingly, the main difference does not seem to lie in the contrast between malleable and moldable tasks, but in the step from continuous to discrete frequencies. This might call for a reconsideration of the requirement that the discrete frequency level of a task need not be changed during task execution. [EK15] demonstrate that for tasks with sufficient runtime, a non-existing frequency might be simulated by using each of the two existing surrounding

[2] Crown Scheduling further assumes that a task can only be allocated to p processors where p is a power of 2, but it turned out [Me19] that this restriction only leads to slightly higher energy compared to requiring an integral number of cores.

discrete frequency levels for part of the execution time, with only a small increase in energy consumption.

The remainder of this article is structured as follows. In Section 2, we briefly review background information on energy-efficient scheduling and related work. In Section 3, we describe both schedulers used, present the routine to convert a Sanders Speck schedule for malleable tasks into a schedule for moldable tasks, and indicate how we discretize the frequency levels. In Section 4, we report on the experiments with which we test the above hypothesis, and in Section 5, we summarize and give an outlook to our future work.

2 Background

We consider the problem of scheduling a set of parallelizable tasks $T = \{t_1, \ldots, t_n\}$ to a set of homogeneous processors $P = \{P_1, \ldots, P_p\}$, where we can scale the frequency for each core independently. A task t_j is characterized by its workload τ_j, which might represent the number of processor cycles necessary to execute the task on one core, and its maximum width W_j, determining the maximum number of cores t_j can be executed on concurrently. A task-specific parallel efficiency function $e_j(q), 1 \leq q \leq p$, is defined as the speedup when running the task on q processors, divided by q, thus depending on the number of cores to which the task is allocated.[3]

If a task t_j is allocated to a noninteger number of processors – as is done by the Sanders Speck scheduler, which assumes tasks to be malleable – the expression $w_j + \gamma_j$ gives the total number of processors for t_j, w_j being an integer value and $\gamma_j \in [0, 1)$.

We compute a static schedule, i. e. we schedule prior to the actual execution. The schedule allocates each task to a set of processor cores and assigns an execution frequency and a start date. The schedule must be feasible, i. e. no core is ever allocated to more than one task at the same time. Furthermore, execution of the task set shall terminate before reaching a deadline M.

When a processor core is running at a frequency f, it draws electrical power $P(f)$. In the most general sense, P is non-decreasing in f. Next to the frequency, P depends on a number of other factors, such as the supply voltage, the instruction mix of the currently executed code and the temperature. We assume that voltage is always set to the minimum level possible for each frequency (and nowadays a single voltage level often serves many frequency levels), so the influence of voltage on power consumption is covered by the frequency parameter. The instruction mix is considered to be uniform for all tasks (but might be extended, cf. [HK17, LKK19]), and the temperature is assumed to be controlled to remain constant.

A task with workload τ that runs on q cores has a workload of $\frac{\tau}{q \cdot e(q)}$ on each core. Normally, it is assumed to run at one frequency f for its whole execution on q cores, so that the power

[3] The concept of maximum width is therefore introduced solely for convenience, as one could simply set $e_j(q) = 0$ for $q > W_j$.

consumption remains constant during the execution. The runtime $t(q)$ can be obtained by dividing core workload (number of cycles) by frequency (number of cycles per time unit). The energy consumption can be obtained by multiplying runtime and power consumption. The energy consumption of a schedule is the sum of the tasks' energy consumptions.

We employ a simple energy model, assuming dynamic power to be (proportional to) f^α [SS12]. Here, f denotes the processor's current operating frequency, and α is a constant depending on the actual hardware. To facilitate comparability between the different scheduling approaches, we assume that frequency scaling does not produce any time or energy overhead, and static power consumption as well as idle power are ignored here. The energy consumed by a processor executing a task on frequency f over a period M is thus $f^\alpha \cdot M$.

While literature on task scheduling is vast, two approaches are of particular interest for our current purposes. In [SS12], a static scheduler for malleable tasks is presented, which allows continuous frequency scaling and seeks to minimize energy consumption while meeting a set deadline. The Crown scheduler introduced in [Me15] is a static scheduler for moldable tasks. It also aims for minimization of energy consumption under deadline constraints, which is achieved via integer linear programming (ILP). Processor allocation, mapping, and frequency scaling are either performed separately or in a combined manner, promising further energy savings. Due to restrictions regarding allocation, mapping, and execution sequence, Crown scheduling allows for solving medium-sized problems by means of linear programming. The Crown scheduling technique is expanded in [MKK16] when static and idle power are taken into consideration and the concept of *core consolidation* is explored. In [XKD12], parallel tasks with deadlines and discrete frequencies are treated. Scheduling is performed via a *level-packing* approach, and for minimization of energy consumption, a 0-1 ILP is contrasted with a three-step heuristic consisting of the specification of each task's width, task scheduling, and frequency assignment.

3 Scheduling Approaches for Parallelizable Tasks

In this section, Sanders Speck scheduling is outlined in 3.1. Subsection 3.2 shows how to convert a Sanders Speck schedule, and 3.3 offers a recap of the crown scheduling technique.

3.1 Sanders Speck Scheduling

Sanders and Speck [SS12] assume that the tasks are malleable, i. e. that the scheduler can vary the number of cores used during execution of a task. For example, a task might be run on 4 cores in the time interval $[0; 0.3 \cdot M]$ and run on 5 cores in the time interval $[0.3 \cdot M; M]$. The number of allocated cores (also called the width) for task j is therefore[4]

[4] They prove that in their setting, other variations than using q and $q + 1$ cores for a task do not occur.

given as $w_j + \gamma_j$, where $w_j = 4$ and $\gamma_j = 0.3$ for this example. There are restrictions on the efficiency function, which according to their analysis seem to be met by many parallel algorithms.

The cores can be scaled to an arbitrary, continuous frequency $f \geq 0$, and the power consumption of a core is f^α, where typically $2 \leq \alpha \leq 3$. Adding a constant amount of static power is ignored, as it does not change the allocations that achieve minimum energy, as long as the cores are not switched off. Thus, transformations can be used to morph realistic frequency ranges into the dimensionless space used here and morph back to power consumption values for real processor architectures.

The algorithm given by Sanders and Speck computes a processor allocation $w_j + \gamma_j$ for each task j, such that $\sum_j (w_j + \gamma_j) = p$, the total energy spent in the computation is minimum, and the task set is executed until the deadline. From the allocation and the given parallel efficiency function, they are able to derive the operating frequency for each task. The mapping is as follows: a task with $w_j \geq 1$ gets w_j cores for the complete time till the deadline. All the γ_j parts, i. e. the times where a task gets an extra core, and the sequential tasks (where $w_j = 0$) are mapped to the remaining $p - \sum_j w_j$ cores by a wrap-around rule: A core is filled with these tasks. When it is allocated for a fraction δ till the deadline, and $\delta + \gamma_j > 1$, then this task gets $1 - \delta$ of the time on this core, and $\gamma_j + \delta - 1$ on the next core. This leads to another requirement: it must be possible to stop a task and continue it later on a different core.

To illustrate the mapping procedure, Figure 1 provides an example mapping of 3 tasks to 6 cores, with processor allocations of 3.2 (t_1, orange), 0.9 (t_2, green), and 1.9 (t_3, pink). Since $w_j > 1$ for the orange and pink tasks, these receive 3 and 1 cores, respectively, for the whole execution time. The sequentially executed green task is not considered at this point. As there are 6 cores in total, the γ_j are distributed across the remaining 2 cores in the next step. The orange task receives core 5 for 20% of the execution time. The green task, which is allocated 0.9 cores overall, is mapped to core 5 for the remaining 80% of total execution time and – by wrap-around – to core 6 for 10%. As one can gather from Figure 1, we now require preemption. The pink task is run on core 6 after t_2 to reach its total allocation of 1.9 cores. For the orange and pink tasks, we assume malleability since their width changes from 4 to 3 (orange) and 1 to 2 (pink) during the course of their execution.

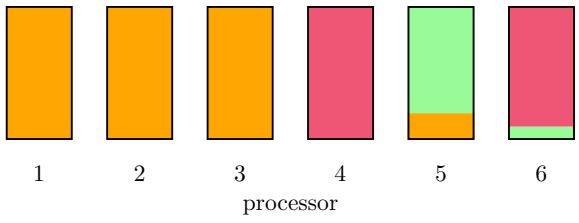


Fig. 1: Example mapping of 3 tasks to 6 cores

3.2 Converting a Sanders Speck Schedule

Converting a Sanders Speck schedule starts with moving from malleable to moldable tasks. Under this restriction, a task can either be executed in parallel on a fixed integer number of processors (i. e. $w_j > 1, \gamma_j = 0.0$), or sequentially on a single core (i. e. $w_j = 1, \gamma_j = 0.0$ or $w_j = 0, 0 < \gamma_j < 1$).

In order to achieve this, one first splits the task set T into $T_P = \{t_j \in T : w_j + \gamma_j > 1\}$ and $T_S = T \setminus T_P$. In the first step, only tasks in T_P are considered. To begin with, we determine the sum of the parallel tasks' processor allocations under the Sanders Speck schedule, $\pi_{total} = \sum_{t_j \in T_P} w_j + \gamma_j$, as well as the number of processors solely dedicated to a single parallel task, $\pi_{single} = \sum_{t_j \in T_P} w_j$. Computing $u = nint(\pi_{total}) - \pi_{single}$ gives you the number of cases in which processor allocation shall be rounded up, where $nint()$ signifies the nearest integer function.[5] One now proceeds as follows: The tasks in T_P are sorted by descending γ_j. Then, for the first u tasks in that order, set $w_j = w_j + 1, \gamma_j = 0.0$. For the remaining $|T_P| - u$ parallel tasks, set $\gamma_j = 0.0$. The processor allocations for parallel tasks now are integer values, and the total amount of cores utilized is $nint(\pi_{total})$. On a side note, it may be the case that a task which would be executed in parallel under the Sanders Speck schedule runs sequentially under the converted schedule.

Going back to the example from Section 3.1, we have $T_P = \{t_1, t_3\}$ and $T_S = \{t_2\}$ since only the green task is executed sequentially. We get $\pi_{total} = w_1 + \gamma_1 + w_3 + \gamma_3 = 3 + 0.2 + 1 + 0.9 = 5.1$ and $\pi_{single} = w_1 + w_3 = 3 + 1 = 4$, which yields $u = nint(\pi_{total}) - \pi_{single} = nint(5.1) - 4 = 5 - 4 = 1$. Sorting the tasks in T_P by descending γ_j gives us t_3, t_1. We now set $w_j = w_j + 1$ and $\gamma_j = 0.0$ for the first u tasks in that order, i. e., for t_3 we assign $w_3 = 2$, $\gamma_3 = 0.0$. For the remaining $|T_P| - u$ parallel tasks, we set $\gamma_j = 0.0$, i. e., for t_1 we now have $\gamma_1 = 0.0$ (and w_1 stays at 3).

The next step is to compute the sequential tasks' processor allocations. The number of cores available for the execution of sequential tasks is $p - nint(\pi_{total})$. The tasks in T_S are now mapped to the remaining processors via a binpacking approach. To this end, they are sorted by descending τ_j and subsequently are assigned to the (at the respective time) least occupied bin, i. e. core.[6] Note that bin size can easily be computed as $f_{max} \cdot M$, which represents the maximum workload a processor can handle up to the deadline M running on its maximum operating frequency f_{max}. After the binning step, one can immediately compute the processor allocation for each task $t_j \in T_S$: If a task t_j is the only one running on a given processor, set $w_j = 1, \gamma_j = 0.0$. Otherwise, a task is allocated a fraction of a

[5] Naturally, u could be computed differently, e. g. one could set $u = \lceil \pi_{total} \rceil - \pi_{single}$ or $u = \lfloor \pi_{total} \rfloor - \pi_{single}$, or try both ways and see which resulting schedule yields lower energy consumption. In any case, as long as there are sequential tasks, it has to be ensured that there is at least one core left for the execution of sequential tasks, i. e. if $\lceil \pi_{total} \rceil = p$, one must set $u = \lfloor \pi_{total} \rfloor - \pi_{single}$.

[6] As the processor's operating frequency for each task is not carried over from the Sanders Speck schedule but is computed anew after processor allocation has been performed, it cannot serve as a sorting criterion. Consequently, a task's workload is used to best represent its size. We expect mapping the sequential tasks to processors in order of descending workload to lead to a reasonable load balancing.

processor corresponding to its share of the processor's total workload, $\gamma_j = \frac{\tau_j}{\sum_{t_k \in T_i} \tau_k}$, where T_i denotes the set of tasks to be executed on processor P_i, and $t_j \in T_i$. For these tasks, set $w_j = 0$.

In our example from Section 3.1, there is just one sequential task and we have $p - nint(\pi_{total}) = 6 - 5 = 1$. Thus, t_2 is allocated an entire core, which gives us $w_2 = 1, \gamma_2 = 0.0$ under the conversion. Figure 2 shows the resulting mapping after conversion. As all parallel tasks' allocations are now integer values, malleability is not required anymore. Beyond that, the mapping of any sequential task to a single core, where it is executed in one go, renders preemption expendable.

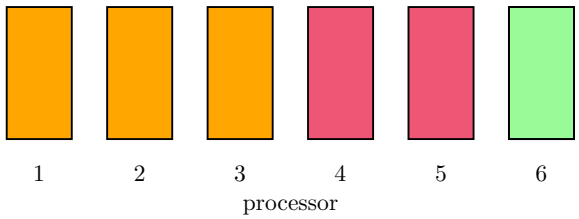


Fig. 2: Mapping after applying the conversion procedure to the example depicted in Figure 1

As processor allocation is now completed for all $t_j \in T$, one can compute the frequency for each task as

$$F_j = \begin{cases} \frac{\tau_j}{\gamma_j \cdot M} & \text{if } w_j = 0, \\ \frac{\tau_j}{e_j(w_j) \cdot w_j \cdot M} & \text{if } w_j \geq 1. \end{cases}$$

This allows calculation of the energy consumption for each $t_j \in T$ (note that $w_j = 0 \Leftrightarrow \gamma_j \neq 0$):

$$E_j = F_j^{\alpha} \cdot (w_j + \gamma_j) \cdot M.$$

Now that processor allocation has been carried out, frequency discretization can be employed in order to obtain energy consumption values under the further restriction that cores feature a set of discrete frequency levels $F = \{f_1, \ldots, f_s\}$ they can run on. We further assume that a core's operating frequency can be changed only between task executions. As before, we first consider T_P: Here, frequency discretization is fairly easy to perform: Since each $t_j \in T_P$ is the only task allocated to its respective processor(s), one has no choice but to increase frequency to the closest frequency level: $F_j = \min\{f \in F : f > F_j\}$. Lowering F_j to the closest frequency level $f < F_j$ instead would incur a deadline violation.

For T_S, frequency discretization does not necessarily imply increasing the operating frequency for each $t_j \in T_S$ to the next possible value. On the contrary, one should aim for reducing the frequency for as many tasks as possible so as to improve energy consumption. To facilitate this, in a first step all $t_j \in T_S$ are treated as described above for parallel tasks: F_j is increased to $\min\{f \in F : f > F_j\}$. Afterwards, for each processor P_i executing tasks

from T_S, the operating frequency is decreased by one level, task by task, until the deadline cannot be met, starting with the last task mapped to P_i (which is the least bulky one due to the binpacking performed beforehand). The last assignment of frequencies to tasks adhering to the deadline is then adopted.

If one allows frequency scaling during task execution, energy consumption can be reduced since a given frequency F_j can be simulated by running on $f_{l_j} = \max\{f \in F : f < F_j\}$ for $c \cdot M$ and on $f_{h_j} = \min\{f \in F : f > F_j\}$ for $(1-c) \cdot M$, $c \in [0,1]$. This procedure forms a generalization of the above approach, where frequency scaling is performed this way for all $t \in T_P$ with $c = 0$. Choosing c individually for all $t \in T$ on the other hand will most likely have a positive impact on energy consumption. This is done as follows:

$$c_j = -\frac{F_j - f_{h_j}}{f_{h_j} - f_{l_j}}.$$

The calculation of a task's energy consumption in this scenario is then pretty straightforward:

$$E_j = (f_{l_j}^{\alpha} \cdot c_j + f_{h_j}^{\alpha} \cdot (1 - c_j)) \cdot (w_j + \gamma_j) \cdot M.$$

3.3 Crown Scheduling

A different static scheduling approach for moldable tasks with discrete frequencies is the *crown scheduler* [Ke13, MKK16]. Here, we consider the *integrated* version, where processor allocation, mapping, and frequency scaling is performed in a combined fashion by solving an integer linear program (ILP). In order to ease computation, the P_i are assigned to a hierarchy of processor groups as in Figure 3: The largest group, P^1, comprises all processors, which is then divided into two equally sized subgroups. These subgroups each are divided into two equally sized subgroups, and so on, with the smallest groups containing one processor only. This group structure can be applied for core counts which are powers of 2, and the resulting number of groups is $2p - 1$. Tasks are then mapped to processor groups, which run at a specified frequency for the duration of a task's execution.

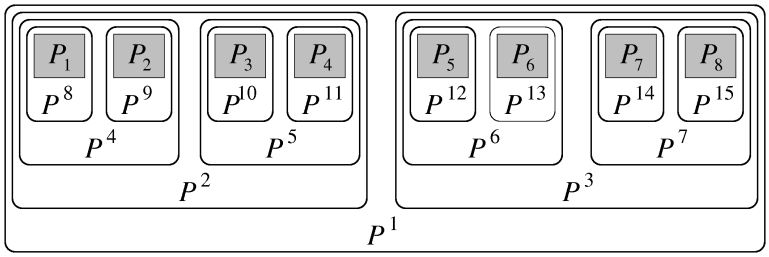


Fig. 3: Processor groups for $p = 8$, taken from [Ke13]

The optimization problem therefore yields $n \cdot (2p-1) \cdot s$ binary decision variables $x_{j,i,k}$, $x_{j,i,k} = 1$ signifying that task t_j shall be executed on processor group P^i operating at frequency f_k. The target function to be minimized computes the total energy consumption

$$E = \sum_{j,i,k} x_{j,i,k} \cdot \frac{\tau_j \cdot f_k^{\alpha-1}}{e_j(p_i)}.$$

The term p_i denotes the number of cores in processor group P^i. Several constraints apply in order to guarantee a valid schedule. First, every task shall be scheduled only once:

$$\forall j : \sum_{i,k} x_{j,i,k} = 1.$$

Furthermore, to ensure the deadline is met, the total runtime of all tasks mapped to a core P_l must not exceed the deadline:

$$\forall l : \sum_{j} \sum_{i \in G_l} \sum_{k} x_{j,i,k} \cdot \frac{\tau_j}{p_i \cdot f_k \cdot e_j(p_i)} \leq M,$$

where G_l denotes the set of all groups core P_l belongs to. Finally, the maximum number of cores allocated to a task t_j shall be its maximum width W_j:

$$\forall j : \sum_{i, p_i > W_j} \sum_{k} x_{j,i,k} = 0.$$

Applying an ILP solver to the optimization problem then yields a mapping of tasks to processor groups as well as the respective processors' operating frequencies.

4 Experiments

In this section, we compare the resulting energy consumption when scheduling task sets via the various approaches presented in Section 3. Our experiments are based on synthetic task sets of varying cardinality and tasks' maximum widths as in [Ke13, MKK16]: A task set comprises 10, 20, 40, or 80 tasks and displays a low ($W_j \in \{1, \ldots, p/2\}$), average ($W_j \in \{p/4, \ldots, 3p/4\}$), high ($W_j \in \{p/2, \ldots, p\}$), or maximum ($\forall j W_j = p$) degree of parallelism[7], or it contains sequential tasks only ($\forall j W_j = 1$). For each combination of cardinality and degree of parallelism, 10 different task sets are considered, thus yielding a total of 200 different task sets for the evaluation of the previously introduced scheduling techniques.

[7] All W_j for low, average, and high degrees of parallelism are determined randomly based on a uniform distribution.

The number of cores is set to 32, and the set of discrete frequencies is $\{1.0, 2.0, 3.0, 4.0, 5.0\}$. Furthermore, $\alpha = 3.0$ and the parallel efficiency of task t_j is defined as in [Ke13]:

$$e_j(q) = \begin{cases} 1 & \text{for } q = 1, \\ 1 - 0.3\frac{q^2}{(W_j)^2} & \text{for } 1 < q \leq W_j, \\ 0.000001 & \text{for } q > W_j. \end{cases}$$

The parameter q is the number of cores t_j is executed on. The deadline M is determined as in [Ke13]:

$$M = \frac{\sum_j \frac{\tau_j}{p \cdot f_{max}} + 2 \sum_j \frac{\tau_j}{p \cdot f_{min}}}{2},$$

where f_{min} and f_{max} are the processors' minimum and maximum operating frequencies.

Energy consumption values are computed for (cf. Figure 4 for the visualization of results):

- the Sanders Speck schedule (reference),
- the Sanders Speck schedule converted to moldable tasks without preemption (blue bar),
- the converted schedule with discrete frequencies (purple bar),
- the converted schedule with discrete frequencies and one-time frequency scaling during task execution (pink bar),
- the crown schedule (yellow bar).

We deployed a C implementation of the Sanders Speck scheduler, a Python implementation of the converter tool, and for crown scheduling, the Gurobi 8.1.0 solver was adopted in conjunction with the gurobipy module for Python. The Sanders Speck scheduler as well as the converter tool were executed sequentially on an AMD Ryzen 7 2700X, while the Gurobi solver ran in 16 threads on 8 cores with a 5 minute timeout for each ILP. It took the Sanders Speck scheduler ≈ 0.3 s to compute schedules for the 200 synthetic task sets, and the conversion process lasted another ≈ 56 s, while the crown scheduler required ≈ 366 min to deliver the results.[8] It should be noted though that the time to solve the ILPs varied heavily among the task sets: Roughly half the schedules could be computed in < 1 s, $\approx 87\%$ in < 10 s, $\approx 92\%$ in < 1 min, and 4 ILPs could not be solved to optimality until the 5 minute timeout occurred.

From Figure 4 it becomes clear that moving from malleable to moldable tasks has a minor impact on energy consumption over all task set types and cardinalities. Subsequent frequency

[8] The execution times given are the sums of user and system times, while the timeout refers to real (wall clock) time.

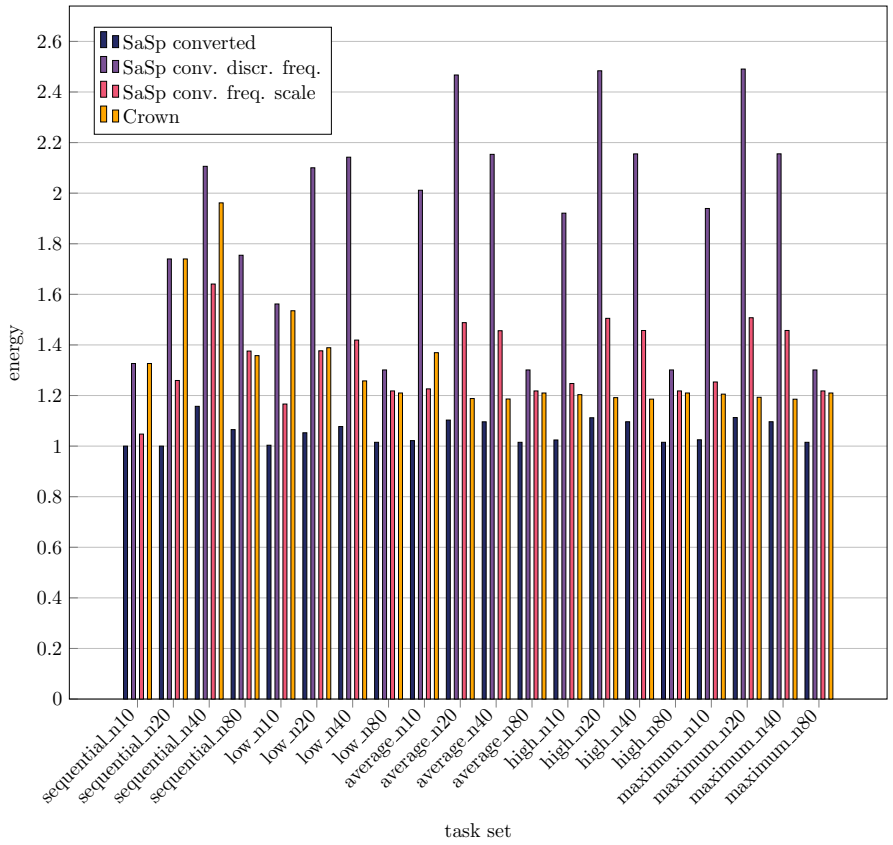


Fig. 4: Energy consumption for different scheduling techniques and synthetic task sets relative to energy consumption for the Sanders Speck scheduler

discretization on the other hand leads to a considerable increase in energy consumption, up to 2.5x compared to the energy consumption of a corresponding Sanders Speck schedule. The effect of frequency discretization is particularly severe for high degrees of parallelism. This is due to the discretization strategy, which permits scaling to lower frequencies for sequentially executed tasks but enforces moving to a higher frequency for tasks running in parallel. The significantly lower relative energy consumption values for high cardinality task sets over all degrees of parallelism support this explanation, since despite their potential for parallel execution, many tasks will have to be run sequentially due to the large number of tasks.

It can also be observed that in many cases the converted Sanders Speck schedule with discretized frequencies leads to a substantial growth in energy consumption over a crown schedule – the exceptions being high cardinality and low maximum width task sets. This is

unfortunate, as both the converter and the crown scheduler essentially operate based on the same constraints. In some scenarios, the former might be preferable though, as producing a converted Sanders Speck schedule is less computationally intensive than creating a crown schedule.

A good deal of the negative effect of frequency discretization can be mitigated when frequency scaling during task execution is permitted. In many cases, the resulting schedule performs better or as good as a crown schedule. It must be pointed out though that the additional overhead of frequency scaling is not reflected in the current findings.

As a final observation, which is not afforded by Figure 4, the absolute energy consumption values hardly differ when the degree of parallelism exceeds the average category. This applies to all scheduling techniques. Here, a preliminary conjecture would be that deadline constraints prevent exploitation of the higher potential for parallelism.

5 Conclusions

We have presented a tool that converts a schedule for malleable tasks on a machine with continuous frequency scaling into one for moldable tasks on a machine with discrete frequency levels. By applying this converter to schedules computed by the Sanders Speck scheduler, we could demonstrate the tradeoff between scheduling time (lower for converted Sanders Speck schedules) and energy-efficiency (better for crown schedules). The average scheduling times are 0.28 s vs. 110 s, while the average energy consumption is $\approx 28\%$ higher for converted Sanders Speck schedules. By doing the conversion in two steps, we see that the crucial point is not the switch from malleable to moldable tasks, but the change from continuous to discrete frequency levels. Hence it might be worthwhile to investigate in future research the influence of frequency switch during task execution, which would allow to "simulate" a continuous frequency f by running a task partly on surrounding discrete frequency levels f_1 and f_2 with $f_1 < f < f_2$.

Acknowledgments

We are very grateful to Christoph Kessler for inspiring our line of research and providing helpful comments.

References

[EK15] Eitschberger, Patrick; Keller, Jörg: Energy-Efficient Task Scheduling in Manycore Processors with Frequency Scaling Overhead. In: Proc. 23rd Euromicro Int. Conf. Parallel, Distributed, and Network-Based Processing (PDP 2015). pp. 541–548, 2015.

[HK17] Holmbacka, Simon; Keller, Jörg: Workload Type-Aware Scheduling on big.LITTLE Platforms. In (Ibrahim, Shadi; Choo, Kim-Kwang Raymond; Yan, Zheng; Pedrycz, Witold, eds): Algorithms and Architectures for Parallel Processing. Springer International Publishing, Cham, pp. 3–17, 2017.

[Ke13] Kessler, Christoph W.; Melot, Nicolas; Eitschberger, Patrick; Keller, Jörg: Crown scheduling: Energy-efficient resource allocation, mapping and discrete frequency scaling for collections of malleable streaming tasks. In: 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation. pp. 215–222, 2013.

[LKK19] Litzinger, S.; Keller, J.; Kessler, C.: Scheduling Moldable Parallel Streaming Tasks on Heterogeneous Platforms with Frequency Scaling. In: Proc. 27th European Signal Processing Conference (EUSIPCO 2019). To appear September 2019.

[Me15] Melot, Nicolas; Kessler, Christoph; Keller, Jörg; Eitschberger, Patrick: Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Manycore Systems. ACM Trans. Archit. Code Optim., 11(4):62:1–62:24, 2015.

[Me19] Melot, Nicolas; Kessler, Christoph; Eitschberger, Patrick; Keller, Jörg: Co-optimizing Core Allocation, Mapping and DVFS in Streaming Programs with Moldable Tasks for Energy Efficient Execution on Manycore Architectures. In: Proc. 19th International Conference on Application of Concurrency to System Design (ACSD 2019). To appear 2019.

[MKK16] Melot, Nicolas; Kessler, Christoph W.; Keller, Jörg: Improving Energy-Efficiency of Static Schedules by Core Consolidation and Switching Off Unused Cores. In: Parallel Computing: On the Road to Exascale (Proc. ParCo 2015). pp. 285–294, 2016.

[SS12] Sanders, Peter; Speck, Jochen: Energy Efficient Frequency Scaling and Scheduling for Malleable Tasks. In (Kaklamanis, Christos; Papatheodorou, Theodore; Spirakis, Paul G., eds): Euro-Par 2012 Parallel Processing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 167–178, 2012.

[XKD12] Xu, H.; Kong, F.; Deng, Q.: Energy Minimizing for Parallel Real-Time Tasks Based on Level-Packing. In: 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications. pp. 98–103, 2012.

B. Juurlink, W. Karl (Hrsg.)
Proceedings 28th PARS Workshop

Enabling Malleability for Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics using LAIK

Amir Raoofy[1], Dai Yang[1], Josef Weidendorfer[2], Carsten Trinitis[1], Martin Schulz[1]

Abstract: Malleability, i.e., the ability for an application to release or acquire resources at runtime, has many benefits for current and future HPC systems. Implementing such functionality, however, is already difficult in newly written code and an even more daunting challenge when considering the enhancement of existing legacy code to support malleability. LAIK is a recent proposal for a dynamic and flexible parallel programming model that separates data and execution into two orthogonal concerns. These properties promise easier malleability as the runtime can partition resources dynamically as needed, as well as easier incremental porting of existing MPI code. In this paper, we explore the malleability of LAIK with the help of `laik-lulesh`, a LAIK-based port of LULESH, a proxy application from the CORAL benchmark suite. We show the steps required for porting the application to LAIK, and we present detailed scaling experiments that show promising results.

Keywords: LAIK, LULESH, MPI, Malleable Application, SPMD

1 Introduction

With the Sierra and Summit systems at LLNL and ORNL, respectively, High Performance Computing (HPC) has reached its last milestone before exascale computing. On this road, the environment of HPC systems has become more and more dynamic; most existing HPC applications, however, are rigid and lack support for malleability. In order to cope with the needed goals in efficiency, energy consumption, and fault tolerance [Ke11], we must overcome such rigidness and enhance HPC applications with more flexibility.

Much effort has been put into enabling malleability in HPC using a wide range of approaches. Some of them target application transparency. An example is MPI Sessions [Ho16], a proposed extension to the MPI standard that allows the instantiation of MPI multiple times at runtime. Other studies focus on the enhancement of HPC system software: Flux [Ah18] is a next-generation job scheduler that allows users to allocate and deallocate resources dynamically in a fine-grained way. Some of the existing work also focuses on the combination of minimal application modification and system software: Invasive MPI

[1] Technical University of Munich, Chair of Computer Architecture and Parallel Systems, Boltzmannstr. 3, 85748 Garching, Germany, {raoofy, yang, trinitis, schulzm}@in.tum.de
[2] Leibniz-Rechenzentrum, Boltzmannstr. 1, 85748 Garching, Germany, josef.weidendorfer@lrz.de

and Invasive Resource Manager [Ur12] provide a modified MPI and a modified SLURM resource manager, which allows for different phases in applications demanding different types and amounts of resources.

A pure application-oriented approach is LAIK [WYT17], a library assisting in dynamically scheduling the execution of HPC applications by separating the concerns for data location and computation. Initially designed for fault tolerance purposes, LAIK also enables elasticity that can be controlled from the outside. It supports incremental porting of an existing application, allowing the user to reuse most of the existing codebases. In this paper, we demonstrate the malleability properties of LAIK on a well-known proxy application - LULESH; an iterative solver for the Sedov Blast Problem, which is highly relevant in real life. LULESH is part of the CORAL benchmarks[3]. It was ported to different parallel programming models [Ka13] for investigation, which allows us to make interesting comparisons with our results.

The main contributions of this paper are: (1) we provide a fully functional port (`laik-lulesh`) of LULESH to LAIK with enhanced malleability features and; (2) we identify the limitations of LAIK with an in-depth performance analysis of our `laik-lulesh` application.

2 Related Work

2.1 LULESH Ports LULESH - as one of the Coral benchmarks - has been ported to a number of programming models and languages, including OpenMP, CUDA, AMP, OpenACC, Loci, Liszt, Chapel, and Charm++. A summary of all versions of LULESH is presented by Karlin et at. [Ka12]. An in-depth study of some ports of LULESH and their performance evaluation is presented by Karlin et at. [Ka13]. According to the authors, `Loci` [LG05] and `Charm++` provide comparable performance to the reference code using MPI.

2.2 Programming Models Many programming models support writing malleable applications. `Charm++`[KK93] is a machine independent parallel programming system. Its dynamic load balancing distributes workloads between different machines at runtime. Adaptive MPI (AMPI) [HLK04] is a flexible MPI implementation based on `Charm++`, with MPI ranks running on virtual Processing Units (PU). The mapping between physical and virtual PUs is done by the `Charm Runtime System (RTS)`, which benefits from the flexibility of Charm++. `Legion` [Ba12] is a data-centric programming model that provides automatic mechanisms for data handling and processing. Based on user-specified workflow mechanisms, the runtime takes care of all data movements during execution. The main difference between `Legion` and `LAIK` is that LAIK works with index spaces and partitionings abstracting data distribution, and it informs the user when changes need to be applied. The physical distribution and processing of the data still rely on the user's specification. Other task-based programming models such as OmpSs[Ma15] and StarPU [Au11] also provide fine granular control of resource and data mapping and processing. With a task-based programming model,

[3] `https://asc.llnl.gov/CORAL-benchmarks/`

malleability is ensured as both workload and data can be easily migrated over hardware resources across task boundaries.

2.3 Benchmarks Many benchmarks are used for performance evaluation of HPC libraries and programming models, e.g., the NAS Parallel Benchmarks[Ba91], the CORAL Benchmarks, and the Rodinia Benchmarks [Ch09].

3 LAIK - Library for Automatic Data Migration

LAIK[WYT17] is a lightweight library for automatic data migration in parallel applications featuring an SPMD approach similar to the one used in many MPI codes. The application programmer is required to transfer the responsibility of partitioning his or her application data to LAIK. However, the actual partitioning algorithm, called a **Partitioner**, which assigns portions of an abstract index space to processes, remains under application programmer control. The application programmer can specify a customized *Partitioner* using callbacks.

Examples provided as presets are the "all" partitioning (the whole index space is replicated to all processes, i.e., requesting complete local copies) or "disjunctive block distribution" (every participating process holds a portion of the index space).

The communication for data structures is implicitly specified as a set of transitions between different partitionings. A transition allows users to specify complex combinations of copy operations, broadcasts and reductions. Transitions can be declared in advance and result in a sequence of abstract communication actions on the index space, to be executed later by the user or by LAIK. The latter is done when asking LAIK to manage the data that is bound to an index space. Executing a transition for such data results in direct communication as required. We can use this approach in LAIK to react to internal and external requests to change the current partitioning by calculating a new partitioning, even on a modified process group, allowing the application to become malleable.

LAIK features several different API-levels, which provide different levels of abstraction, allowing programmers to port their application incrementally. The basic "index space" API only transfers the responsibility of index space partitioning to LAIK. The corresponding transitions and action sequences are calculated by LAIK, while the actual communication is carried out by the user application. The more sophisticated "data" API allows developers to hand over complete control of data structures to LAIK, allowing automatic communication. This way, all communication is hidden from the user, resulting in lean, purly data-oriented code. In addition, LAIK does internal optimization for different communication patterns, which reduces development costs.

Previous studies with LAIK [WYT17; Ya18] have already shown the effectiveness and efficiency of LAIK in both performance and basic malleability, but are limited to simple data structures with simple communication patterns. In this paper, we evaluate LAIK's

capability by porting the rather complex program LULESH to LAIK. For our experiments, we use the published open-source LAIK-Version on Github[4].

4 Porting The Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) to LAIK

Livermore Unstructured Lagrange Explicit Shock Hydrodynamics (LULESH) is a benchmark that solves the Sedov blast problem using Lagrangian hydrodynamics [HKG].

In each time step of the simulation, a number of hydrodynamic fields are updated by the computational kernels. Data is stored either related to the initially cube-shaped finite elements covering a 3d domain ('elemental' data) or related to the vertices of the finite elements ('nodal' data). After the initialization of the fields, coordinates and boundary conditions the timestep loop executes until convergence.

For our work, we use the MPI/OpenMP hybrid implementation of LULESH 2.0[5], where MPI communication is explicitly coded. This is typical for many HPC applications: developers do not want to start from scratch for an existing application to add new functionality such as malleability.

The selected implementation of LULESH is based on domain decomposition for data partitioning and uses non-blocking communication (`Isend/Irecv`) at various points in each time step. Two main kernels that require communication are executed in each iteration: (1) stencil-wise updates of data structures such as velocity gradients which require halo exchange at borders of domains. (2) aggregation of contributions from element quantities to the surrounding nodes, e.g., in the calculation of force fields, which also requires a halo exchange followed by an aggregation. Our goal in porting LULESH to LAIK is to keep the number of changes as small as possible: code accessing data structures as well as computational kernels should remain unchanged. LAIK is used for two tasks: a) it is responsible for regular value updates in iterations (e.g., for halo exchanges), replacing MPI in the original code; b) it has to migrate data for re-distribution to support malleability. Correspondingly, porting can be done in two steps: first, we replicate original communication by letting LAIK maintain the data structures that get updated in each iteration. Second, for malleability, also data structures used purely locally have to be maintained by LAIK, as it also needs to be migrated on re-distribution. Furthermore, small modifications are required in the main iteration loop to check for re-partitioning requests and trigger data repartitioning in LAIK. It is important to mention that *LULESH 2.0 only supports a cubic number of MPI processes*. This limitation still holds for our LAIK implementation as we are neither changing the partitioning algorithm nor the compute kernel. In the following, we present the steps for porting LULESH to LAIK. The major changes made to the LULESH program execution flow are illustrated in Pseudocode 1 vs. Pseudocode 2.

[4] `https://github.com/envelope-project/laik`, commit e504385
[5] The base version for our port is `https://github.com/LLNL/LULESH`, commit a328f79.

Pseudocode 1: Simplified Pseudocode for MPI Implementation of LULESH [HKG]

```
MPI_Init();
Domain locDom ← InitMeshDecomposition(rank, size, sz);
while (!endOfSimulation) do
    CalcTimeIncrement();
    LagrangeLeapFrog();
end
MPI_Finalize();
```

Pseudocode 2: Simplified Pseudocode for `laik-lulesh` [HKG]

```
Laik_init_mpi();
Domain locDom ← InitMeshDecomposition(rank, size, sz);
while (!endOfSimulation) do
    if (needRepart) then
        Domain newDom ← InitMeshDecomposition(newRank, newSize, sz);
        Laik_repartitioning_and_migrate(locDom, newDom);
        locDom ← newDom;
    end
    CalcTimeIncrement();
    LagrangeLeapFrog();
end
Laik_finalize();
```

Step 1: Adaptation of data structures requiring MPI communication. LULESH uses asynchronous communication for force fields, namely `fx`, `fy`, `fz` and `nodalMass` followed by aggregation. LAIK provides such communication patterns through so-called "transitions" between different partitionings, which are trigged by calling the API call `laik_switch`. For that, we create halo regions by introducing *overlapping* partitions on a global nodal index space, so that the neighboring tasks share one layer of nodes. As LAIK does not provide such partitioning out of the box, we implement the partitioner algorithm ourselves as a core part of our `laik-lulesh` port. While this partitioner is similar to the one in the reference code, it uses a different layout: the reference code uses `std::vector` with a compact `xyz` (Figure 1 right) layout, while our implementation of `laik-lulesh` relies on a non-compact `xyz` (Figure 1 left) layout. It divides a local domain into "slices". The reason for this layout is that, although LULESH works on a 3D domain, it is mapped into a 1D data structure during execution. As we stick to the original kernels, we also need to provide 1D data storage.

From the perspective of LAIK this data is shared between the two neighboring tasks and updated by each task independently. After each iteration, we call `laik_switch`, which triggers the reduction operations on shared data, replacing any explicit communication code. LULESH uses an asynchronous halo exchange for velocity gradient fields, i.e., `delv_xi`, `delv_eta` and `delv_zeta`, and these data structures needed to be ported to LAIK as well. For

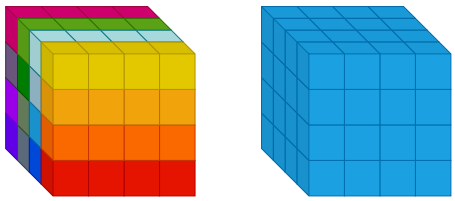


Fig. 1: Illustration of `laik_slices` for a problem with local domains of size $4 \times 4 \times 4$ elements. Each color represents one `laik_slice` in the partitioning. Current implementation of `laik_lulesh` relies on many slices (left) and the reference code relies on a compact allocation of data (right).

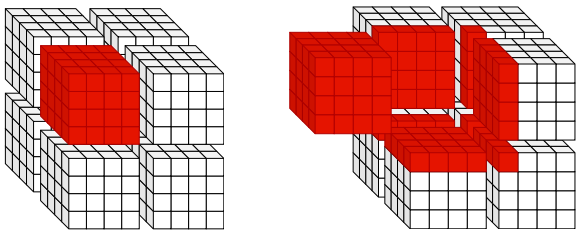


Fig. 2: Element partitionings: *exclusive* partitioning (left) and *halo* (right) partitioning. This figure is an illustration for a problem with $8 \times 8 \times 8$ elements using 8 tasks (8 sub-domains, e.g., cubes to be processes by each task). Each sub-domain (indicated in red) contains $4 \times 4 \times 4$ elements in exclusive `partitioning` and halo `partitioning` extends it with one layer.

this, we create two partitionings — *exclusive* and *halo* — again using custom partitioners. A switch between them triggers the communications for the halo exchange (see Figure 2).

Internally, LULESH uses `std::vector` as its data container and implements an accessor interface layer on top. This allows modifications of the underlying data structure by only replacing `std::vector`. For that, we introduce `laik_vector` encapsulating LAIK structures and implement the required accessor interfaces. In the end, all data structures with communication requirements are ported to LAIK and all MPI calls are eliminated.

Step 2: Enabling Malleability. LAIK can support malleability by having control over the underlying data structures. As LULESH uses a number of data structures in addition to those mentioned above, those need to be handed over to LAIK as well. For that, we provide additional partitions according to the needed data distribution before and after repartitioning. As above, calling `laik_switch` triggers the required MPI communication under the hood and re-distributes data according to the target partitioning. In addition, we modify the main loop to handle repartitioning requests. If a process is no longer part of an active calculation after repartitioning, this process is discarded by calling `laik_finalize`.

Additional Optimizations. We consider multiple optimizations to improve the performance of `laik-lulesh`. First, the transitions are executed in every iteration. Therefore, we use LAIK's advanced APIs in order to pre-calculate and cache of the `transitions` and

corresponding `action_sequences` between the above-mentioned partitionings. Moreover, our implementation creates many `1D` slices and, as referencing data between `laik_switches` normally invalidates the pointers for all the slices, we use LAIK's reservation API, which guarantees the validity of addresses of data across "switch"es.

5 Evaluation

To show the performance of our ported LULESH code, we carried out strong and weak scaling tests on SuperMUC Phase II (SuperMUC) which consists of 3072 nodes, each equipped with 2 Intel Haswell Xeon Processor E5-2697v3 and 64 GB of main memory.

5.1 Weak Scaling We execute both the reference code `lulesh2.0` and our ported `laik-lulesh` five times with problem size 16^3 corresponding to $s = 16$ (parameter `-s`). We report the average runtime per iteration without initialization and finalization. The upper bound of the number of iteration is 10,000. The normalized runtime per iteration is represented in Figure 3 by the box plots and noted on the y-axis. On the x-axis, the number of MPI Tasks used in experiments is given. We can see an increase in iteration runtime with an increasing number of MPI tasks from our code. However, the reference code scales almost perfectly with only a slight increase. The blue line in Figure 3 represents overhead, which scales up with the number of processes. Our hypothesis for the source of this increasing overhead is the lack of support for asynchronous communication in LAIK and we, therefore, continue with strong scaling experiments to pinpoint the source of this overhead.

The overhead in our `laik-lulesh` is in an acceptable range for a mid-sized run (e.g., 10% at 512 processes), which is a realistic use case scenario for **a malleable application**. For extreme scaling, however, LAIK must be further adapted and tuned.

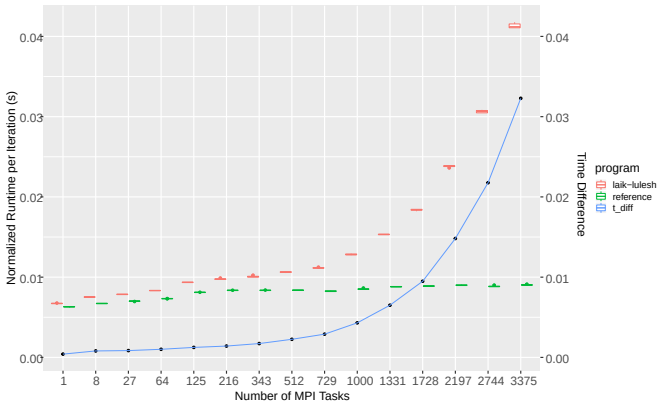


Fig. 3: Weak scaling comparison of `laik-lulesh` with reference LULESH

5.2 Strong Scaling Due to the limited support for an only cubic number of processes, the following limitation applies: let $C = s^3 * p$ be the global 3-dimensional problem size to be

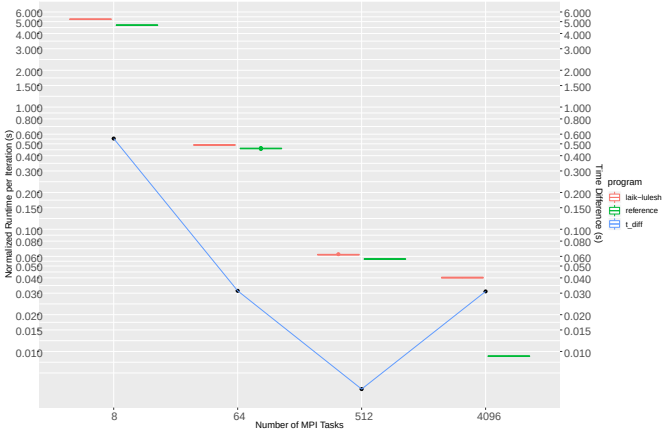


Fig. 4: Strong Scaling Comparison of `laik-lulesh` with reference LULESH

held constant for all strong scaling experiments, with s being the local one-dimensional problem size (parameter `-s` i); furthermore, let p be the number of MPI processes and $S = C^{\frac{1}{3}}$; the following implication applies: $(C = s^3 * p) => (S = s * p^{\frac{1}{3}})$. As $p^{\frac{1}{3}}$ and s must be natural numbers we set up our strong scaling experiments with $p^{\frac{1}{3}}$ being powers of 2 and $S = 256$. The resulting corresponding tuples of (p, s) used in this paper for strong scaling are $(1^3 = 1, 256)$, $(2^3 = 8, 128)$, $(4^3 = 64, 64)$, $(8^3 = 512, 32)$ and $(16^3 = 4096, 16)$. The results from these experiments are illustrated in Figure 4. Note that the y-axis is log(2)-scaled. As expected, similar scaling behavior can be observed for both the LAIK version and the reference version with up to 512 processes. With 4096 processes, our port shows significant overhead (factor 2x slower than the reference code). In addition, the overhead curve first decreases then increases with a large number of processes. Figure 4 shows a relatively constant overhead for experiments with 8, 64, and 512 processes. This is most likely due to a constant overhead of using 1D slices in the LAIK implementation. In addition, Figure 4 shows a problem similar to weak scaling for `laik-lulesh` with a large number of processes. This is very likely the result of lack of support for asynchronous communication in LAIK, which scales with the number of point-to-point communication (and the number of processes).

5.3 Repartitioning Using LAIK, we can now shrink the number of MPI processes during the execution of `laik-lulesh`. To test how this shrinking affects the scaling behavior before and after repartitioning, we conduct a series of scaling experiments. We set up the repartitioning experiments with $p^{\frac{1}{3}}$ being powers of 2 and $S = 64$ and enforce a repartitioning to the smaller, next supported number of MPI processes in the strong scaling tests. This results in the following repartitioning experiments: from 8 to 1, from 64 to 8 and from 512 to 64. We fix the number of iterations (parameter `-i`) to 2000 iterations for all experiments and execute the kernel for 250 iterations with the initial number of MPI processes and then perform the

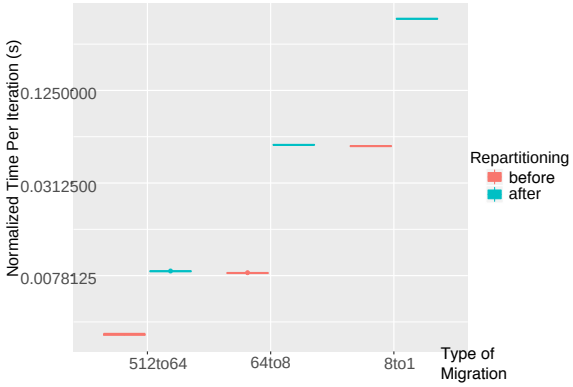


Fig. 5: Strong scaling result for `laik-lulesh` with enforced repartitioning

repartitioning. Finally, the kernel is executed another 1750 iterations until completion. We execute a total of five runs per configuration on SuperMUC.

The results are provided in Figure 5. On the x-axis, we list the type of migration, and on the y-axis the normalized time per iteration in log scale, respectively. Both curves show the same trend. In addition, the runtime for a given scale (e.g., p=64) is almost the same, regardless of whether it is the initial number of processes or the state after repartitioning. The required time for repartitioning is presented in Table 1.

As for the effectiveness of the repartitioning function, our test shows little to no overhead on the normalized kernel execution time of `laik-lulesh` and also a migration has little impact.

Tab. 1: Required Time for `laik-lulesh` with Enforced Repartitioning

Configuration	Time for Repartitioning
512to64	~1.5678s
64to8	~0.8803s
8to1	~1.6979s

6 Conclusions and Future Work

We presented `laik-lulesh`, a port of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) to LAIK. The ported code gains enhanced malleability at runtime. Moreover, it is capable of repartitioning its data as needed. All data structures, as well as all MPI communication, are transferred to LAIK's responsibility, while the actual kernel did not have to be modified. Results from weak and strong scaling experiments show a low constant overhead and an increasing overhead when scaling the number of processes. The constant part of the overhead stems from the additional abstraction introduced by LAIK. The overall overhead stays acceptable for up to 1000 MPI processes, which is a

typical configuration for malleable applications. Further, the repartitioning experiments show promising performance result, as no additional overhead from repartitioning can be observed. This shows that LAIK is a useful approach to assist programmers to enable malleability for existing HPC applications.

As next steps, we will enhance LAIK's asynchronous communication behavior. In addition, we will work on the reduction of the constant overhead by using a proposed layout interface from LAIK. Finally, we plan to work with the original LULESH team to overcome the limitation of only allowing a cubic number of processes.

Acknowledgment This work is partially funded by the German Federal Ministry for Education and Research under grant title 01|H16010D (Project ENVELOPE). Compute resources on SuperMUC are sponsored by Leibniz Supercomputer Centre under grant title pr27ne.

References

[Ah18] Ahn, D. H.; Bass, N.; Chu, A.; Garlick, J.; Grondona, M.; Herbein, S.; Koning, J.; Patki, T.; Scogland, T. R. W.; Springmeyer, B.; Taufer, M.: Flux: Overcoming Scheduling Challenges for Exascale Workflows. In: 2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS). Pp. 10–19, Nov. 2018.

[Au11] Augonnet, C.; Thibault, S.; Namyst, R.; Wacrenier, P.-A.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience 23/2, pp. 187–198, 2011.

[Ba12] Bauer, M.; Treichler, S.; Slaughter, E.; Aiken, A.: Legion: Expressing locality and independence with logical regions. In: High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for. IEEE, pp. 1–11, 2012.

[Ba91] Bailey, D. H.; Barszcz, E.; Barton, J. T.; Browning, D. S.; Carter, R. L.; Dagum, L.; Fatoohi, R. A.; Frederickson, P. O.; Lasinski, T. A.; Schreiber, R. S., et al.: The NAS parallel benchmarks. The International Journal of Supercomputing Applications 5/3, pp. 63–73, 1991.

[Ch09] Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J. W.; Lee, S.-H.; Skadron, K.: Rodinia: A benchmark suite for heterogeneous computing. In: Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on. IEEE, pp. 44–54, 2009.

[HKG] Hornung, R. D.; Keasler, J. A.; Gokhale, M. B.: Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory, tech. rep. LLNL-TR-490254, Livermore, CA, USA, pp. 1–17.

[HLK04] Huang, C.; Lawlor, O.; Kalé, L. V.: Adaptive MPI. In (Rauchwerger, L., ed.): Languages and Compilers for Parallel Computing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 306–322, 2004.

[Ho16] Holmes, D.; Mohror, K.; Grant, R. E.; Skjellum, A.; Schulz, M.; Bland, W.; Squyres, J. M.: MPI Sessions: Leveraging Runtime Infrastructure to Increase Scalability of Applications at Exascale. In: Proceedings of the 23rd European MPI Users' Group Meeting. ACM, pp. 121–129, 2016.

[Ka12] Karlin, I.; Bhatele, A.; Chamberlain, B. L.; Cohen, J.; Devito, Z.; Gokhale, M.; Haque, R.; Hornung, R.; Keasler, J.; Laney, D.; Luke, E.; Lloyd, S.; McGraw, J.; Neely, R.; Richards, D.; Schulz, M.; Still, C. H.; Wang, F.; Wong, D.: LULESH Programming Model and Performance Ports Overview, tech. rep. LLNL-TR-608824, Livermore, CA, USA, Dec. 2012, pp. 1–17.

[Ka13] Karlin, I.; Bhatele, A.; Keasler, J.; Chamberlain, B. L.; Cohen, J.; Devito, Z.; Haque, R.; Laney, D.; Luke, E.; Wang, F., et al.: Exploring traditional and emerging parallel programming models using a proxy application. In: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing. IEEE, pp. 919–932, 2013.

[Ke11] Kerbyson, D.; Vishnu, A.; Barker, K.; Hoisie, A.: Codesign Challenges for Exascale Systems: Performance, Power, and Reliability. Computer 44/11, pp. 37–43, Nov. 2011.

[KK93] Kale, L. V.; Krishnan, S.: CHARM++: A Portable Concurrent Object Oriented System Based on C++. In: Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications. OOPSLA '93, ACM, Washington, D.C., USA, pp. 91–108, 1993.

[LG05] Luke, E. A.; George, T.: Loci: A rule-based framework for parallel multi-disciplinary simulation synthesis. Journal of Functional Programming 15/3, pp. 477–502, 2005.

[Ma15] Martsinkevich, T.; Subasi, O.; Unsal, O.; Cappello, F.; Labarta, J.: Fault-Tolerant Protocol for Hybrid Task-Parallel Message-Passing Applications. In: 2015 IEEE International Conference on Cluster Computing. Pp. 563–570, Sept. 2015.

[Ur12] Ureña, I. A. C.; Riepen, M.; Konow, M.; Gerndt, M.: Invasive MPI on Intel's Single-Chip Cloud Computer. In (Herkersdorf, A.; Römer, K.; Brinkschulte, U., eds.): Architecture of Computing Systems – ARCS 2012. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 74–85, 2012.

[WYT17] Weidendorfer, J.; Yang, D.; Trinitis, C.: LAIK: A Library for Fault Tolerant Distribution of Global Data for Parallel Applications. In: Konferenzband des PARS'17 Workshops. Hagen, Germany, 2017.

[Ya18] Yang, D.; Weidendorfer, J.; Kuestner, T.; Trinitis, C.; Ziegler, S.: Enabling Application-Integrated Proactive Fault Tolerance. In. Vol. 32, IOS Press, Bologna, Italy, pp. 475–484, 2018.

B. Juurlink, W.Karl (Hrsg.)
Proceedings 28th PARS Workshop

Comparing MPI Passive Target Synchronization Schemes on a Non-Cache-Coherent Shared-Memory Processor

Steffen C hristgau[1], Bettina Schnor[2]

Abstract: MPI passive target synchronisation offers exclusive and shared locks. These are the building blocks for the implementation of applications with Readers & Writers semantic, like for example distributed hash tables. This paper discusses the implementation of MPI passive target synchronisation on a non-cache-coherent multicore, the Intel Single-Chip Cloud Computer. The considered algorithms differ in their communication style, their data structures, and their semantics. It is shown that shared memory approaches scale very well and deliver good performance, even in absence of cache coherence.

Keywords: process synchronization; programming models and systems for manycores; MPI

1 Introduction

Distributed hash tables (DHTs) are a common approach for fast data access in big data and data analytics applications. However, DHTs imply dynamic communication which makes an implementation using two-sided communication, i. e. with `SEND` and `RECV` operations, cumbersome. In contrast, one-sided communication (OSC) with `PUT` and `GET` operations is a suited programming model for a DHT with its dynamic communication pattern.

Concerning the process coordination, a DHT application follows the Readers & Writers model [CS17a]: reads may occur concurrently while inserts have to be done exclusively. Hence, a resource has to be locked before it is updated. Typically, writers are given preference to avoid readers reading old data. This coordination scheme maps on MPI's *passive* target synchronization which offers *exclusive locks* (one writer) and *shared locks* (many readers). In addition, an MPI implementation has much freedom to implement the process synchronization for passive target OSC [Me15, p. 448].

This paper discusses different synchronisation algorithms on the experimental non-cache-coherent 48-core Intel Single-Chip Cloud Computer (SCC) [Ho10]. Figure 1 shows an architectural overview of the chip. While core counts steadily increase, the management of cache coherence becomes a more challenging task due to that high number of cores and high memory bandwidths [Mo15]. Although coherent high-end processors with 64 cores are currently available, non-coherent architectures provide an interesting research domain.

[1] Zuse Insitute Berlin, Supercomputing Department, christgau@zib.de
[2] University of Potsdam, Institute for Computer Science, schnor@cs.uni-potsdam.de

It has been shown in previous work that such nCC shared-memory systems can be easily programmed with well established technologies like, e. g., MPI [CS17b].

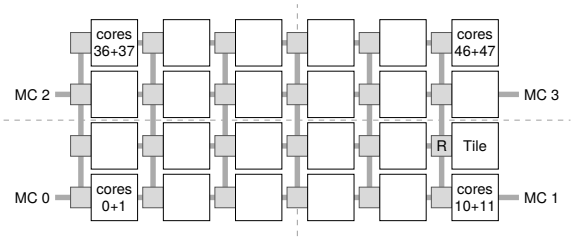


Fig. 1: Overview of the Intel SCC.

In this paper we compare the performance of three synchronization schemes for MPI passive target OSC: the message-based scheme from MPICH, the writer preference locks by Mellor-Crummey and Scott [MS91a] for shared memory systems (MCS-WP), and a best effort approach originally designed for RMA-capable distributed memory systems by Gerstenberger, Besta and Hoefler (GBH) [GBH14].

The next section gives an overview over related work. The different synchronization schemes and their data structures are described in Section 3, their implementation on the SCC is described in Section 4. Results from a micro-benchmarks are presented in Section 5, followed by a discussion. Section 6 concludes the paper.

2 Related Work

An early work on the topic of efficient MPI OSC implementations is the discussion for InfiniBand clusters [Ji04]. Recently, implementation schemes for NUMA-aware locks on cache-coherent multicore machines are gaining interest [DMS15, GLQ16, KMK17, CFMC15], but non-cache-coherent architectures are still neglected.

Concerning the SCC, the authors of [AMB12] investigate barrier synchronization on the SCC and use the Message Passing Buffer (MPB) to store the synchronization data. In [ASB14], they even exploit unused entries in the rare lookup tables of the chip's memory subsystem. The bottom line of this research is that synchronization data should be placed close to the spinning core. RCKMPI [UGT12] is a tuned message-based MPI implementation for the SCC and uses the fast on-chip MPB for message transport. One-sided communication is fully supported but is based on messages as well. In case of MPI's *general active synchronization*, we have already shown that an implementation using shared memory and uncached memory accesses outperforms the message-based approach significantly [CS17b]. Similar, Reble et al. discuss the active target *fence synchronization* style which they implement on top of an efficient barrier [RCL13].

Regarding Distributed Memory Architectures, Gerstenberger et al. have published performance numbers of a distributed hash table application running on up to 32k cores [GBH14].

They use their own MPI-3.0 RMA library implementation for Cray Gemini and Aries interconnects called foMPI (fast one-sided MPI). The presented synchronization scheme for passive synchronization is described in Section 3.1 and adapted for the SCC (see Section 4). Schmid et al. have proposed a scheme for Readers & Writers locking dedicated for distributed memory architectures with RMA capabilities like the Cray XC30 [SBH16]. The synchronization data structures are organized hierarchically in a distributed tree.

Subsuming the related work, there are no efforts in *passive target synchronization* for nCC many-core CPUs with shared memory like the SCC.

3 Synchronization Schemes for nCC Architectures

This section describes three implementation designs for MPI passive target synchronization. The first two are known from the literature. The third one describes the default implementation on the SCC. While [GBH14] presents a best-effort approach for a distributed-memory machine, the work from [MS91a] addresses scaling on shared-memory machines.

3.1 GBH Best Effort Synchronization

In [GBH14], Gerstenberger, Besta and Hoefler (GBH) present an implementation for MPI passive target synchronization for the Cray XC super-computers. It is based on atomic remote direct memory operations (RDMA) operations which are supported by the hardware. The design uses two stages of counters for each created window object: a single global counter and per-process local counters. All counters are allocated in memory close to the owning process. The global counter resides in the memory of a designated process (rank 0). All counters are accessible by RDMA operations.

The global counter tracks active `LOCKALL` operations and exclusive locks which are mutual exclusive. The per-process counter indicates the number of active exclusive and shared locks. As there can be only one exclusive access at a given time and process, a single bit is used to indicate such epochs (see Fig. 2).

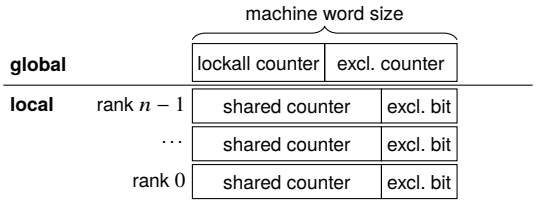


Fig. 2: Counters used by the GBH synchronization scheme.

Whenever a lock of either type is going to be acquired, the respective fields in the counters are incremented using atomic operations which return the previous value (fetch and add).

When a conflict is detected, e. g. shared locks are active at a process but an exclusive one should be acquired, the counter modifications are reverted and the process tries again at a later time using an exponentially growing back-off. The scheme does not distinguish between different process types, so any reader may overtake writers.

3.2 MCS Locks with Writer Preference

To avoid centralized spin objects which cause high interconnect traffic, Mellor-Crummey and Scott proposed MCS locks [MS91a]. Those are based on linked lists of lock objects that are allocated in shared memory. Each process that wants to enter a critical section by means of MCS locks appends a list entry which consists of a boolean flag `blocked` and a pointer to the next waiting process. The flag is initially set to `TRUE`. A process that wants to acquire the lock repeatedly polls the flag in its list entry until it is set to `FALSE` by a process which releases its lock.

The main advantage of using one list item per process is that spinning is done only on a local list item and not on a globally shared one like a single spin lock, for example.

Based on the original MCS locks, which do not differentiate between process types, Mellor-Crummey et al. present specialized locks that give precedence to either reading or writing processes [MS91b]. We have implemented MCS locks with writer preference (MCS-WP), since it fits best to the DHT use case where lots of readers and rare writers are expected.

Independent of the precedence, the proposed lock data structures contain lists for waiting reader and writer processes as shown in Figure 3. In addition to the lists, there is a `state` variable which is a single integer variable. For writer-precedence, the state tracks the number of active readers and provides flags for indicating presence of interested readers, interested writers, and active writers. Those are manipulated with atomic operations [MS91b, Sec. 3]. For usage with MPI passive target synchronization, every window i is associated with a lock data structure L_i as shown in Figure 3.

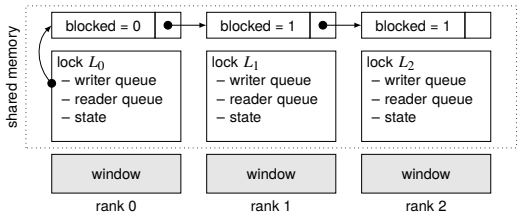


Fig. 3: MCS-lock based data structure for reader or writer precedence.

3.3 Message-Based Synchronization

RCKMPI, the MPI implementation for the SCC, uses messages to implement passive target synchronization. This behaviour is inherited from MPICH's CH3 device implementation but varies depending on the configuration. By default, the `LOCK` synchronization and subsequent communication operations are deferred until the end of the `UNLOCK` operation. The library then sends a control message from the origin to the target process, waits for a reply, i. e. *lock granted message*, issues the communication operations, and signals the unlock operation by setting an according field in the message header of the final communication operation. The unlock indicator can be piggy-backed in case of a single RMA operation. If no RMA operation is performed, no messages are sent. However, MPICH/RCKMPI can be configured to send a control message to the target for lock acquisition at the beginning of the access epoch. This also ensures transfer of control messages even in the absence of communication. Although this implements one-sided communication it actually requires participation of the target to process the synchronization messages.

Independent of the active configuration, the lock requests from different origins are serialized at the target process. Since the received messages are processed in the order at which they are received by the target, there is no preference of readers or writers (or lock type).

4 Implementation on the SCC

The SCC is not a product but a research vehicle [Ho10]. Each of the 48 cores has two integrated 16 KB L1 caches – one for data and instructions – as well as an external unified 256 KB L2 cache. There is no cache coherence between the caches of different cores, but every core can access all memory location. In addition to main memory, a fast 16 kB Message Passing Buffer (MPB) is placed on each tile.

All of the of the above synchronization schemes have been implemented in RCKMPI. Messages exchange is done by writing messages in the receiver's MPB who polls the buffer for new incoming messages. This implementation is considered as the baseline version.

The GBH and MCS-WP implementations do not use messages. Instead, the required data structures are allocated in shared off-chip DRAM memory. Due to the non-coherent architecture of the SCC, those data structures are polled using uncached memory accesses. While previous research proved that polling the on-tile MPB or even the Lookup Tables (see Section 2) reduces the traffic on the interconnect, both approaches are hardly feasible in our case due to a resource conflict (MPB) or a missing resource management (LUTs). Therefore, the external DRAM is used as a resource for the synchronization data. As shown in previous work [CS17a], synchronization data is allocated in a distributed fashion: Per-process data is stored in the DRAM memory close to the owning core.

5 Experimental Evaluation

We evaluate the different design schemes using a communication-free microbenchmark. The experiments were conducted on a SCC system with cores clocked at 533 MHz and 800 MHz for the mesh network and the memory controllers. A total of 32 GB of RAM was installed on the system. Each core runs Linux 3.1.4 with platform-relevant patches applied. Software is cross-compiled using GCC 4.4.6 with optimization (`-O2`), and MPICH 3.1.3 was used as the foundation MPI implementation.

5.1 Microbenchmark description

The employed microbenchmark measures the latency for a pair of LOCK/UNLOCK operations. No communication is performed between those two operations. The time for performing these operations is compared for the GBH and MCS-WP implementations as well as for the message-based but SCC-optimized RCKMPI. Because the default RCKMPI implementation defers the synchronization, we also measure RCKMPI with a forced message exchange for synchronization (cf. Section 3.3).

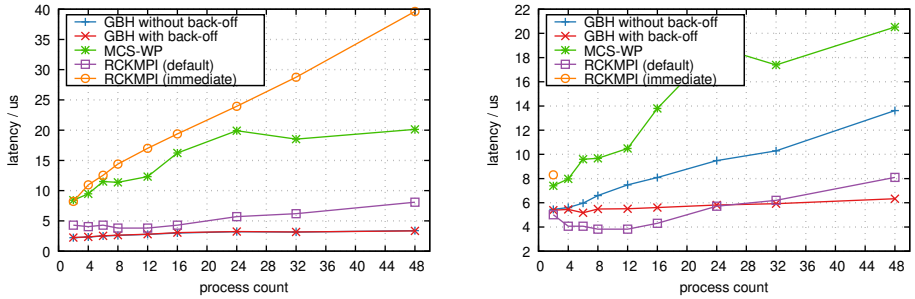
Each process of the microbenchmark performs 1000 pairs of `LOCK`/`UNLOCK` calls in a tight loop. The type of the employed lock is controlled by an input parameter that specifies the share of shared and exclusive locks each process shall issue. According to that parameter, every process randomly decides between the two lock types. The target process is chosen randomly as well and may include the origin process. The access mode (shared or exclusive) will obviously have an influence on the results. Therefore, three different ratios of shared and exclusive locks, i. e.. readers and writers, were measured: only shared locks (only readers), all accesses are made with exclusive locks (only writers) and a mixture of both where shared and exclusive accesses are equally distributed (see Figure 4a–5).

Since we are interested in the scaling of the different synchronization schemes, we run the benchmark with different numbers of processes. The processes are mapped according to the core with matching number. That is, the rows of the SCC's mesh network are filled before moving to the next row. In case for 24 processes, the chip's lower half (see Figure 1) is filled.

From each of the 1000 `LOCK`/`UNLOCK` cycles, the required time is measured. Finally, all samples from all processes are gathered and the median time from all synchronization operations is computed. This value is shown in the following diagrams for different core counts. We compare MCS-WP, RCKMPI with both immediate messaging (synchronization message upon method call) and default behaviour (no messages), and GBH. In addition to GBH, a version without back-off is included in the evaluation in order to analyze the impact of the back-off on the synchronization latency. For the version with back-off, the initial delay between two lock acquisition attempts is 1 µs. This value is doubled for each consecutive failed attempt. It has been shown for the SCC that the usage of back-offs can improve the performance of synchronization primitives [RCL13].

5.2 Results: Shared Locks Only

Figure 4a shows the latency of the different implementations when all accesses are shared. The RCKMPI implementation with immediate messaging has the highest latency due to overhead from sending and processing the control messages. The default RCKMPI implementation includes only library overhead but no message exchange and scales therefore well. It is slightly slower than both GBH versions due to additional management in the message-based code path that are not used for the GBH and MCS implementations.



(a) Shared locks only.

(b) 50% shared and 50% exclusive locks.

Fig. 4: Synchronization latencies for the shared-only and mixed cases.

Both GBH versions exhibit nearly constant and identical synchronization latencies because no conflicts occur in the shared-only use case and thus no back-off is required. Consequently, the two curves overlap in the plot. Similar, the reason for the constant time is that shared accesses are not mutual exclusive. In the GBH scheme, acquiring a shared lock only involves incrementing the shared counter in the target's local counter (see Fig. 2). Due to the distribution of the synchronization data and missing exclusive locks, which might cause more attempts to acquire the shared lock, no contention on these counters is observed on the SCC.

In case of the of the MCS-WP, the latency is generally higher than for GBH. The latter only involves incrementing a single per-process counter value, but for MCS the state variable needs to be checked and list data has to be changed. This causes the operations to take longer than for GBH.

From the data one can also note an increasing latency for up to 24 processes. After that, the latency remains nearly constant with a slight drop for 32 processes. This observation can be attributed to the distributed synchronization data. With up to 24 processes, the two lower memory controllers of the chip (see Figure 1) have to handle the polling requests of the 12 processes associated to each of them. Additional processes are then handled by the next memory controllers, but do not increase the load on the already utilized ones.

This is also the reason for the slight latency drop at 32 processes: Since the upper two memory controllers have to serve fewer processes than the lower two, the median latency

reduces. Similar behavior can be identified for the switch from 6 (only handled by MC 0) to 8 processes (MC 1 handles additional polling accesses).

Since for 24 processes the two lower memory controllers experience maximum usage and because of the distributed data, no further increase of the lock latency is observed when the number of MPI processes is raised. This is different from the statement in [SBH16, p. 11], that MCS locks that distinguish between readers and writers do not scale well under heavy read contention. We are not able to confirm this remark by our experiments on the SCC.

5.3 Results: Lock Type Mix

In Figure 4b, the results for the 50% mix of shared and exclusive locks is displayed. For this workload, no data — except for two processes — could be acquired for the immediate RCKMPI variant. The benchmark deadlocked in those cases. Our assumption is that required responses to control message are not sent when they are expected. This might be due to absent message processing and might be solved by triggering process through `MPI_Test` calls. However, a deeper investigation was out of the paper's scope.

The default variant of RCKMPI which does not send any message unsurprisingly performs as in case for shared lock.

For GBH, the latency is slightly increased compared to the previous results. The scaling, however, remains nearly identical and still shows a constant time for the synchronization for all process counts. The increased latency can be accounted to the higher probability for an unsuccessful attempt for lock acquisition. In such a case, the processes perform their back-off but are able to acquire the lock in a later attempt very soon, since the median latency only increases by about 3 µs.

Opposite to GBH with back-off, the version without this feature shows a latency that increases linear with the number of processes. The effect is due to the contention. This can be explained by a competition for both the global and the per-process counter variables. This reduces the chance of a lock acquisition for either process type. Especially, the global counter must be modified both at the beginning and at the end of the lock attempt — notably, this has to be done also in the unsuccessful case. Since the global counter is a centralized data structure, contention on the responsible memory controller is likely.

With the exception of the GBH without back-off and the dysfunctional immediate RCKMPI version, the overall performance and scaling is identical to the previous scenario.

For MCS-WP, an almost identical performance as in the previous experiment is observed. While two different process types are active, the same data structures are used and the same operations (state manipulation and list management) are performed. Thus, the overall performance stays the same.

5.4 Results: Exclusive Locks Only

Finally, Figure 5 shows the scaling where only exclusive locks are used. The GBH variant without back-off clearly suffers from the sole usage of exclusive locks and its aggressive best-effort approach. The effect of contention on the global counter from the previous experiment is amplified which causes increased latency.

Contrary to that observation, the other synchronization schemes still perform with identical scaling behavior and similar absolute latency. For GBH with back-off, the latency increases slightly and approaches MCS-WP. This might be caused by an increased number of attempts to acquire the lock.

For MCS-WP, the performance is still equivalent to the previous experiments. Because writers just queue up at the individual per-process queues (cf. Figure 3), the median time to acquire the lock does not increase. Further, the completely distributed data structures pays off as no contention occurs.

The immediate RCKMPI variant works without problems in this experiments. However, the latency is up to about four times higher than for the other implementations. Moreover, linear scaling can be observed.

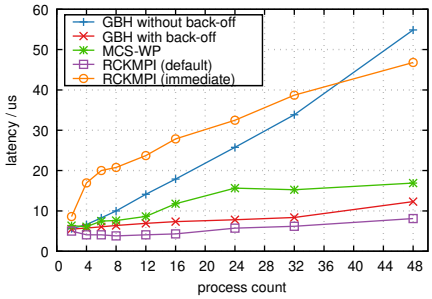


Fig. 5: Latency for exclusive locks.

5.5 Discussion

Although a tuned implementation for message transfer is available on the SCC, it does not pay off in case for MPI passive target synchronization. Besides issues with deadlocks, which may be fixable, the observed latency is much higher than for the presented memory-based approaches that use uncached-memory accesses due to the immanent data transfer and processing overhead.

Contrary, the memory-based schemes perform with low latencies and nearly constant scaling and are therefore a favorable choice for nCC shared memory architectures like the SCC. The implemented synchronization schemes take special care for the distribution of data

structures and their access pattern. The MCS-WP scheme avoids centralized data, and the GBH scheme with back-off uses a rate-limited access to its data structures.

6 Conclusion

In this paper, we discussed and evaluated three different synchronization schemes for non-cache-coherent shared memory architectures, like the SCC. Two memory-based schemes known from the literature have been implemented for that platform. The evaluation shows that such schemes are well-suited for nCC many-core architectures both in terms of absolute performance and scalability. Despite, they employ uncached memory operations, the approaches even outperform competitors that rely on SCC-optimized message passing.

The experiments also show that the MCS-WP scheme which gives precedence to writers can be used on nCC systems without scalability or severe performance degradations. The reason is the avoidance of centralized data structures. To achieve a comparable performance, the GBH scheme that also uses centralized data structures in addition to distributed ones, a back-off mechanism appears to be crucial for the median latency. Nevertheless, the algorithms have to be evaluated in the context of an application in subsequent work. Future work may also include an analysis how the presented approaches perform on contemporary processors built from multiple *chiplets* when taking their inherent NUMA-design and hardware support for cache coherence into consideration.

Bibliography

[AMB12] Al-Khalissi, Hayder; Marongiu, Andrea; Berekovic, Mladen: Low-Overhead Barrier Synchronization for OpenMP-like Parallelism on the Single-Chip Cloud Computer. In: Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University, November 29th-30th 2012, Aachen, Germany. 2012.

[ASB14] Al-Khalissi, Hayder; Shah, Syed Abbas Ali; Berekovic, Mladen: An Efficient Barrier Implementation for OpenMP-Like Parallelism on the Intel SCC. In: 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014. 2014.

[CFMC15] Chabbi, Milind; Fagan, Michael; Mellor-Crummey, John: High Performance Locks for Multi-level NUMA Systems. In: Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. PPoPP 2015, ACM, New York, NY, USA, 2015.

[CS17a] Christgau, Steffen; Schnor, Bettina: Design of MPI Passive Target Synchronization for a Non-Cache-Coherent Many-Core Processor. In: Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware: 27. PARS Workshop. volume 34 of Mitteilungen. Gesellschaft für Informatik, 2017.

[CS17b] Christgau, Steffen; Schnor, Bettina: Exploring one-sided communication and synchronization on a non-cache-coherent many-core architecture. Concurrency and Computation: Practice and Experience, 29(15), 2017.

[DMS15] Dice, David; Marathe, Virendra J.; Shavit, Nir: Lock Cohorting: A General Technique for Designing NUMA Locks. ACM Trans. Parallel Comput., 1(2), February 2015.

[GBH14] Gerstenberger, Robert; Besta, Maciej; Hoefler, Torsten: Enabling highly-scalable remote memory access programming with MPI-3 One Sided. Scientific Programming, 22(2), 2014.

[GLQ16] Guiroux, Hugo; Lachaize, Renaud; Quéma, Vivien: Multicore Locks: The Case is Not Closed Yet. In: Proceedings of the 2016 USENIX Conference on Usenix Annual Technic al Conference. USENIX ATC '16, USENIX Association, Berkeley, CA, USA, 2016.

[Ho10] Howard, Jason et al.: A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS. In: Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International. February 2010.

[Ji04] Jiang, Weihang et al.: Efficient Implementation of MPI-2 Passive One-Sided Communication on InfiniBand Clusters. In (Kranzlmüller, Dieter; Kacsuk, Péter; Dongarra, Jack J., eds): 11th European PVM/MPI Users' Group Meeting, Budapest, Hungary. volume 3241 of Lecture Notes in Computer Science. Springer, 2004.

[KMK17] Kashyap, Sanidhya; Min, Changwoo; Kim, Taesoo: Scalable NUMA-aware Blocking Synchronization Primitives. In: Proceedings of the 2017 USENIX Conference on Usenix Annual Techni cal Conference. USENIX ATC '17, USENIX Association, Berkeley, CA, USA, 2017.

[Me15] Message Passing Interface Forum: , MPI: A Message-Passing Interface Standard, Version 3.1. online, June 2015. http://www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf.

[Mo15] Morgan, Timothy Prickett: , More Knights Landing Xeon Phi Secrets Unveiled, March 2015. http://www.nextplatform.com/2015/03/25/more-knights-landing-xeon-phi-secrets-unveiled/ accessed 2019-08-28.

[MS91a] Mellor-Crummey, John M.; Scott, Michael L.: Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors. ACM Trans. Comput. Syst., 9(1), 1991.

[MS91b] Mellor-Crummey, John M.; Scott, Michael L.: Scalable Reader-Writer Synchronization for Shared-Memory Multiprocessors. In (Wise, David S., ed.): Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP), Williamsburg, Virginia, USA, April 21-24, 1991. ACM, 1991.

[RCL13] Reble, Pablo; Clauss, Carsten; Lankes, Stefan: One-sided communication and synchronization for non-coherent memory-coupled cores. In: International Conference on High Performance Computing & Simulation, HPCS 2013. IEEE, 2013.

[SBH16] Schmid, Patrick; Besta, Maciej; Hoefler, Torsten: High-Performance Distributed RMA Locks. In (Nakashima, Hiroshi; Taura, Kenjiro; Lange, Jack, eds): Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 - June 04, 2016. ACM, 2016.

[UGT12] Ureña, Isaías A. Comprés; Gerndt, Michael; Trinitis, Carsten: Wait-Free Message Passing Protocol for Non-coherent Shared Memory Architectures. In: 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria. 2012.

1. Aktuelle und zukünftige Aktivitäten (Bericht des Sprechers)

Die 35. Ausgabe der PARS-Mitteilungen enthält die Beiträge des 28. PARS-Workshops.

Der 28. PARS-Workshop fand am 20. und 21. März 2019 an der Technischen Universität Berlin statt. Der Workshop war mit 30 Teilnehmern gut besucht. Am Morgen des ersten Tages hielten Nadjib Mammeri und Sohan Lal von der TU Berlin (Embedded System Architecture) ein Tutorial zum Thema „LPGPU2“. Insgesamt wurden 11 sehr interessante Vorträge präsentiert, die zusammen ein umfangreiches Themenspektrum abdeckten. Prof. Dr.-Ing. Jeronimo Castrillon (TU Dresden) hielt einen eingeladenen Vortrag zum Thema „Programming abstractions: When domain-specific goes mainstream“.

Auch dieses Mal wurde wieder der mit 500 EUR dotierte Nachwuchspreis vergeben. Er ging in diesem Jahr an Farzaneh Salehiminapour (TU Berlin). Die Übergabe erfolgte am Ende des PARS Workshops. Professor Ben Juurlink und Daniel Maier (TU Berlin) sei herzlich für die reibungslose Organisation gedankt.

Preisübergabe in der Halle der TU Berlin
Prof. Dr. Wolfgang Karl mit der Preisträgerin Farzaneh Salehiminapour.

Während des PARS-Workshops fand auch eine Sitzung des PARS-Leitungsgremiums statt. In das Leitungsgremium wurden als neue Mitglieder Dr. Steffen Christgau (ZIB Berlin), Dr. Stefan Lankes (RWTH Aachen) und Prof. Dr. Martin Schulz (TU München) aufgenommen.

Vom 10. bis 11. Oktober 2019 fand in Hünfeld ein Perspektiv-Workshop des Leitungsgremiums der FG PARS statt.

Unser nächster Workshop ist der

14. PASA-Workshop am 23. Und 24. Mai 20120 in Aachen.

Der Workshop wird wie in den vergangenen „geraden“ Jahren gemeinsam mit der Fachgruppe ALGO im Rahmen der Tagung ARCS 2020 durchgeführt.

Aktuelle Informationen finden Sie auch auf der PARS-Webpage

http://fg-pars.gi.de/

Die PARS-Mitteilungen und ihre Beiträge erscheinen auch in der Digitalen Bibliothek der Gesellschaft für Informatik (https://dl.gi.de/handle/20.500.12116/1903).

Anregungen und Beiträge für die Mitteilungen können an den Sprecher (wolfgang.karl@kit.edu) gesendet werden.

Ich wünsche allen ein gutes und erfolgreiches Jahr 2020.

Karlsruhe im Januar 2020

Prof. Dr. Wolfgang Karl

2. Zur Historie von PARS

Bereits am Rande der Tagung CONPAR81 vom 10. bis 12. Juni 1981 in Nürnberg wurde von Teilnehmern dieser ersten CONPAR-Veranstaltung die Gründung eines Arbeitskreises im Rahmen der GI: Parallel-Algorithmen und -Rechnerstrukturen angeregt. Daraufhin erfolgte im Heft 2, 1982 der GI-Mitteilungen ein Aufruf zur Mitarbeit. Dort wurden auch die Themen und Schwerpunkte genannt:

1) **Entwurf von Algorithmen für**
   - verschiedene Strukturen (z. B. für Vektorprozessoren, systolische Arrays oder Zellprozessoren)
   - Verifikation
   - Komplexitätsfragen
2) **Strukturen und Funktionen**
   - Klassifikationen
   - dynamische/rekonfigurierbare Systeme
   - Vektor/Pipeline-Prozessoren und Multiprozessoren
   - Assoziative Prozessoren
   - Datenflussrechner
   - Reduktionsrechner (demand driven)
   - Zellulare und systolische Systeme
   - Spezialrechner, z. B. Baumrechner und Datenbank-Prozessoren
3) **Intra-Kommunikation**
   - Speicherorganisation
   - Verbindungsnetzwerke
4) **Wechselwirkung zwischen paralleler Struktur und Systemsoftware**
   - Betriebssysteme
   - Compiler
5) **Sprachen**
   - Erweiterungen (z. B. für Vektor/Pipeline-Prozessoren)
   - (automatische) Parallelisierung sequentieller Algorithmen
   - originär parallele Sprachen
   - Compiler
6) **Modellierung, Leistungsanalyse und Bewertung**
   - theoretische Basis (z. B. Q-Theorie)
   - Methodik
   - Kriterien (bezüglich Strukturen)
   - Analytik

In der Sitzung des Fachbereichs 3 ‚Architektur und Betrieb von Rechensystemen' der Gesellschaft für Informatik am 22. Februar 1983 wurde der Arbeitskreis offiziell gegründet. Nachdem die Mitgliederzahl schnell anwuchs, wurde in der Sitzung des Fachausschusses 3.1 ‚Systemarchitektur' am 20. September 1985 in Wien der ursprüngliche Arbeitskreis in die Fachgruppe FG 3.1.2 ‚Parallel- Algorithmen und -Rechnerstrukturen' umgewandelt.

Während eines Workshops vom 12. bis 16. Juni 1989 in Rurberg (Aachen) - veranstaltet von den Herren Ecker (TU Clausthal) und Lange (TU Hamburg-Harburg) - wurde vereinbart, Folgeveranstaltungen hierzu künftig im Rahmen von PARS durchzuführen.

Beim Workshop in Arnoldshain sprachen sich die PARS-Mitglieder und die ITG-Vertreter dafür aus, die Zusammenarbeit fortzusetzen und zu verstärken. Am Dienstag, dem 20. März 1990 fand deshalb in

München eine Vorbesprechung zur Gründung einer gemeinsamen Fachgruppe PARS statt.

Am 6. Mai 1991 wurde in einer weiteren Besprechung eine Vereinbarung zwischen GI und ITG sowie eine Vereinbarung und eine Ordnung für die gemeinsame Fachgruppe PARS formuliert und den beiden Gesellschaften zugeleitet. Die GI hat dem bereits 1991 und die ITG am 26. Februar 1992 zugestimmt.

3. Bisherige Aktivitäten

Die PARS-Gruppe hat in den vergangenen Jahren mehr als 20 Workshops durchgeführt mit Berichten und Diskussionen zum genannten Themenkreis aus den Hochschulinstituten, Großforschungseinrichtungen und der einschlägigen Industrie. Die Industrie - sowohl die Anbieter von Systemen wie auch die Anwender mit speziellen Problemen - in die wissenschaftliche Erörterung einzubeziehen war von Anfang an ein besonderes Anliegen. Durch die immer schneller wachsende Zahl von Anbietern paralleler Systeme wird sich die Mitgliederzahl auch aus diesem Kreis weiter vergrößern.

Neben diesen Workshops hat die PARS-Gruppe die örtlichen Tagungsleitungen der CONPAR-Veranstaltungen:

CONPAR 86 in Aachen,
CONPAR 88 in Manchester,
CONPAR 90 / VAPP IV in Zürich und
CONPAR 92 / VAPP V in Lyon
CONPAR 94/VAPP VI in Linz

wesentlich unterstützt. In einer Sitzung am 15. Juni 1993 in München wurde eine Zusammenlegung der Parallelrechner-Tagungen von CONPAR/VAPP und PARLE zur neuen Tagungsserie EURO-PAR vereinbart, die vom 29. bis 31. August 1995 erstmals stattfand:

Euro-Par'95 in Stockholm

Zu diesem Zweck wurde ein „Steering Committee" ernannt, das europaweit in Koordination mit ähnlichen Aktivitäten anderer Gruppierungen Parallelrechner-Tagungen planen und durchführen wird. Dem Steering Committee steht ein „Advisory Board" mit Personen zur Seite, die sich in diesem Bereich besonders engagieren. Die offizielle Homepage von Euro-Par ist **http://www.europar.org/**.

Außerdem war die Fachgruppe bemüht, mit anderen Fachgruppen der Gesellschaft für Informatik übergreifende Themen gemeinsam zu behandeln: Workshops in Bad Honnef 1988, Dagstuhl 1992 und Bad Honnef 1996 (je zusammen mit der FG 2.1.4 der GI), in Stuttgart (zusammen mit dem Institut für Mikroelektronik) und die PASA-Workshop-Reihe 1991 in Paderborn, 1993 in Bonn, 1996 in Jülich, 1999 in Jena, 2002 in Karlsruhe, 2004 in Augsburg, 2006 in Frankfurt a. Main und 2008 in Dresden (jeweils gemeinsam mit der GI-Fachgruppe 0.1.3 ‚Parallele und verteilte Algorithmen (PARVA)') sowie 2012 in München, 2014 in Lübeck, 2016 in Nürnberg und 2018 in Braunschweig (gemeinsam mit der GI-Fachgruppe ALGO, die Nachfolgegruppe von PARVA). Der nächste PASA Workshop wird wieder gemeinsam mit der GI FG ALGO 2020 in Aachen stattfinden.

PARS-Mitteilungen/Workshops:

Aufruf zur Mitarbeit, April 1983 (Mitteilungen Nr. 1)

Erlangen, 12./13. April 1984 (Mitteilungen Nr. 2)

Braunschweig, 21./22. März 1985 (Mitteilungen Nr. 3)

Jülich, 2./3. April 1987 (Mitteilungen Nr. 4)

Bad Honnef, 16.-18. Mai 1988 (Mitteilungen Nr. 5, gemeinsam mit der GI-Fachgruppe 2.1.4 'Alternative Konzepte für Sprachen und Rechner')

München Neu-Perlach, 10.-12. April 1989 (Mitteilungen Nr. 6)

Arnoldshain (Taunus), 25./26. Januar 1990 (Mitteilungen Nr. 7)

Stuttgart, 23./24. September 1991, "Verbindungsnetzwerke für Parallelrechner und Breitband-Übermittlungssysteme" (Als Mitteilungen Nr. 8 geplant, gem. mit ITG-FA 4.1, 4.4 und GI/ITG FG Rechnernetze, wg. Kosten nicht erschienen. siehe Tagungsband Inst. für Mikroelektronik Stuttgart.)

Paderborn, 7./8. Oktober 1991, "Parallele Systeme und Algorithmen" (Mitteilungen Nr. 9, 2. PASA-Workshop)

Dagstuhl, 26.-28. Februar 1992, "Parallelrechner und Programmiersprachen" (Mitteilungen Nr. 10, gemeinsam mit der GI-Fachgruppe 2.1.4 'Alternative Konzepte für Sprachen und Rechner')

Bonn, 1./2. April 1993, "Parallele Systeme und Algorithmen" (Mitteilungen Nr. 11, 3. PASA-Workshop)

Dresden, 6.-8. April 1993, "Feinkörnige und Massive Parallelität" (Mitteilungen Nr. 12, zusammen mit PARCELLA)

Potsdam, 19./20. September 1994 (Mitteilungen Nr. 13, Parcella fand dort anschließend statt)

Stuttgart, 9.-11. Oktober 1995 (Mitteilungen Nr. 14)

Jülich, 10.-12. April 1996, "Parallel Systems and Algorithms" (4. PASA-Workshop), Tagungsband erschienen bei World Scientific 1997)

Bad Honnef, 13.-15. Mai 1996, zusammen mit der GI-Fachgruppe 2.1.4 'Alternative Konzepte für Sprachen und Rechner' (Mitteilungen Nr. 15)

Rostock, (Warnemünde) 11. September 1997 (Mitteilungen Nr. 16, im Rahmen der ARCS'97 vom 8.-11. September 1997)

Karlsruhe, 16.-17. September 1998 (Mitteilungen Nr. 17)

Jena, 7. September 1999, "Parallele Systeme und Algorithmen" (5. PASA-Workshop im Rahmen der ARCS'99)

An Stelle eines Workshop-Bandes wurde den PARS-Mitgliedern im Januar 2000 das Buch 'SCI: Scalable Coherent Interface, Architecture and Software for High-Performance Compute Clusters', Hermann Hellwagner und Alexander Reinefeld (Eds.) zur Verfügung gestellt.

München, 8.-9. Oktober 2001 (Mitteilungen Nr. 18)

Karlsruhe, 11. April 2002, (Mitteilungen Nr. 19)

Travemünde, 5./6. Juli 2002, Brainstorming Workshop "Future Trends" (Thesen in Mitteilungen Nr. 19)

Basel, 20./21. März 2003 (Mitteilungen Nr. 20)

Augsburg, 26. März 2004 (Mitteilungen Nr. 21)

Lübeck, 23./24. Juni 2005 (Mitteilungen Nr. 22)

Frankfurt/Main, 16. März 2006 (Mitteilungen Nr. 23)

Hamburg, 31. Mai / 1. Juni 2007 (Mitteilungen Nr. 24)

Dresden, 26. Februar 2008 (Mitteilungen Nr. 25)

Parsberg, 4./5. Juni 2009 (Mitteilungen Nr. 26)

Hannover, 23. Februar 2010 (Mitteilungen Nr. 27)

Rüschlikon, 26./27. Mai 2011 (Mitteilungen Nr. 28)

München, 29. Februar 2012 (Mitteilungen Nr. 29)

Erlangen, 11.+12. April 2013 (Mitteilungen Nr. 30)

Lübeck, 25. Februar 2014 (Mitteilungen Nr. 31)

Potsdam, 7.+8. Mai 2015 (Mitteilungen Nr. 32)

Nürnberg, 4.+5. April 2016 (Mitteilungen Nr. 33)

Hagen, 4.+5. Mai 2017 (Mitteilungen Nr. 34)

Berlin, 21. +22. März 2019 (Mitteilungen Nr. 35)

4. Mitteilungen (ISSN 0177-0454)

Mit dieser Ausgabe sind 35 Mitteilungen zur Veröffentlichung der PARS-Aktivitäten und verschiedener Workshops erschienen. Darüberhinaus enthalten die Mitteilungen Kurzberichte der Mitglieder und Hinweise von allgemeinem Interesse, die dem Sprecher zugetragen werden.

Teilen Sie - soweit das nicht schon geschehen ist - Tel., Fax und E-Mail-Adresse der GI-Geschäftsstelle mitgliederservice@gi-ev.de mit für die zentrale Datenerfassung und die regelmäßige Übernahme in die PARS-Mitgliederliste. Das verbessert unsere Kommunikationsmöglichkeiten untereinander wesentlich.

5. Vereinbarung

Die Gesellschaft für Informatik (GI) und die Informationstechnische Gesellschaft im VDE (ITG) vereinbaren die Gründung einer gemeinsamen Fachgruppe

Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware,

die den GI-Fachausschüssen bzw. Fachbereichen:

FA 0.1	Theorie der Parallelverarbeitung
FA 3.1	Systemarchitektur
FB 4	Informationstechnik und technische Nutzung der Informatik

und den ITG-Fachausschüssen:

FA 4.1	Rechner- und Systemarchitektur
FA 4.2/3	System- und Anwendungssoftware

zugeordnet ist.

Die Gründung der gemeinsamen Fachgruppe hat das Ziel,

- die Kräfte beider Gesellschaften auf dem genannten Fachgebiet zusammenzulegen,
- interessierte Fachleute möglichst unmittelbar die Arbeit der Gesellschaften auf diesem Gebiet gestalten zu lassen,
- für die internationale Zusammenarbeit eine deutsche Partnergruppe zu haben.

Die fachliche Zielsetzung der Fachgruppe umfasst alle Formen der Parallelität wie

- Nebenläufigkeit
- Pipelining
- Assoziativität
- Systolik
- Datenfluss
- Reduktion
  etc.

und wird durch die untenstehenden Aspekte und deren vielschichtige Wechselwirkungen umrissen. Dabei wird davon ausgegangen, dass in jedem der angegebenen Bereiche die theoretische Fundierung und Betrachtung der Wechselwirkungen in der Systemarchitektur eingeschlossen ist, so dass ein gesonderter Punkt „Theorie der Parallelverarbeitung“ entfällt.

1. Parallelrechner-Algorithmen und -Anwendungen

   - architekturabhängig, architekturunabhängig
   - numerische und nichtnumerische Algorithmen
   - Spezifikation
   - Verifikation
   - Komplexität
   - Implementierung

2. Parallelrechner-Software

   - Programmiersprachen und ihre Compiler
   - Programmierwerkzeuge
   - Betriebssysteme

3. Parallelrechner-Architekturen

   - Ausführungsmodelle
   - Verbindungsstrukturen
   - Verarbeitungselemente
   - Speicherstrukturen
   - Peripheriestrukturen

4. Parallelrechner-Modellierung, -Leistungsanalyse und -Bewertung

5. Parallelrechner-Klassifikation, Taxonomien

Als Gründungsmitglieder werden bestellt:

von der GI: Prof. Dr. A. Bode, Prof. Dr. W. Gentzsch, R. Kober, Prof. Dr. E. Mayr, Dr. K. D. Reinartz, Prof. Dr. P. P. Spies, Prof. Dr. W. Händler

von der ITG: Prof. Dr. R. Hoffmann, Prof. Dr. P. Müller-Stoy, Dr. T. Schwederski, Prof. Dr. Swoboda, G. Valdorf

Ordnung der Fachgruppe
Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware

1. Die Fachgruppe wird gemeinsam von den Fachausschüssen 0.1, 3.1 sowie dem Fachbereich 4 der Gesellschaft für Informatik (GI) und von den Fachausschüssen 4.1 und 4.2/3 der Informationstechnischen Gesellschaft (ITG) geführt.

2. Der Fachgruppe kann jedes interessierte Mitglied der beteiligten Gesellschaften beitreten. Die Fachgruppe kann in Ausnahmefällen auch fachlich Interessierte aufnehmen, die nicht Mitglied einer der beteiligten Gesellschaften sind. Mitglieder der FG 3.1.2 der GI und der ITG-Fachgruppe 6.1.2 werden automatisch Mitglieder der gemeinsamen Fachgruppe PARS.

3. Die Fachgruppe wird von einem ca. zehnköpfigen Leitungsgremium geleitet, das sich paritätisch aus Mitgliedern der beteiligten Gesellschaften zusammensetzen soll. Für jede Gesellschaft bestimmt deren Fachbereich (FB 3 der GI und FB 4 der ITG) drei Mitglieder des Leitungsgremiums: die übrigen werden durch die Mitglieder der Fachgruppe gewählt. Die Wahl- und die Berufungsvorschläge macht das Leitungsgremium der Fachgruppe. Die Amtszeit der Mitglieder des Leitungsgremiums beträgt vier Jahre. Wiederwahl ist zulässig.

4. Das Leitungsgremium wählt aus seiner Mitte einen Sprecher und dessen Stellvertreter für die Dauer von zwei Jahren; dabei sollen beide Gesellschaften vertreten sein. Wiederwahl ist zulässig. Der Sprecher führt die Geschäfte der Fachgruppe, wobei er an Beschlüsse des Leitungsgremiums gebunden ist. Der Sprecher besorgt die erforderlichen Wahlen und amtiert bis zur Wahl eines neuen Sprechers.

5. Die Fachgruppe handelt im gegenseitigen Einvernehmen mit den genannten Fachausschüssen. Die Fachgruppe informiert die genannten Fachausschüsse rechtzeitig über ihre geplanten Aktivitäten. Ebenso informieren die Fachausschüsse die Fachgruppe und die anderen beteiligten Fachausschüsse über Planungen, die das genannte Fachgebiet betreffen. Die Fachausschüsse unterstützen die Fachgruppe beim Aufbau einer internationalen Zusammenarbeit und stellen ihr in angemessenem Umfang ihre Publikationsmöglichkeiten zur Verfügung. Die Fachgruppe kann keine die Trägergesellschaften verpflichtenden Erklärungen abgeben.

6. Veranstaltungen (Tagungen/Workshops usw.) sollten abwechselnd von den Gesellschaften organisiert werden. Kostengesichtspunkte sind dabei zu berücksichtigen.

7. Veröffentlichungen, die über die Fachgruppenmitteilungen hinausgehen, z. B. Tagungsberichte, sollten in Abstimmung mit den den Gesellschaften verbundenen Verlagen herausgegeben werden. Bei den Veröffentlichungen soll ein durchgehend einheitliches Erscheinungsbild angestrebt werden.

8. Die gemeinsame Fachgruppe kann durch einseitige Erklärung einer der beteiligten Gesellschaften aufgelöst werden. Die Ordnung tritt mit dem Datum der Unterschrift unter die Vereinbarung über die gemeinsame Fachgruppe in Kraft.

CALL FOR PAPERS
14th Workshop on Parallel Systems and Algorithms
PASA 2020

https://www.cs12.tf.fau.de/conf/pasa2020/

in conjunction with
ARCS 2020 – 33rd International Conference on Architecture of Computing Systems
Aachen, Germany, 25 - 26 May 2020

organized by
GI/ITG-Fachgruppe 'Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware' (PARS)
and GI-Fachgruppe 'Algorithmen' (ALGO)

The PASA workshop series has the goal to build a bridge between theory and practice in the area of parallel systems and algorithms. In this context practical problems which require theoretical investigations as well as the applicability of theoretical approaches and results to practice shall be discussed. An important aspect is communication and exchange of experience between various groups working in the area of parallel computing, e.g. in computer science, electrical engineering, physics or mathematics.

Topics of Interest include, but are not restricted to:

- parallel architectures & storage systems
- parallel and distributed algorithms
- parallel embedded systems
- models of parallel computation
- ubiquitous and pervasive systems
- scheduling and load balancing
- reconfigurable parallel computing
- parallel programming languages
- data stream-oriented computing
- software engineering for parallel systems
- interconnection networks
- parallel design patterns
- network and grid computing
- performance evaluation of parallel systems
- distributed and parallel multimedia systems
- parallel algorithms for artificial intelligence applications

PASA 2020 Webpage: **https://www.cs12.tf.fau.de/conf/pasa2020/**

The workshop will comprise invited talks on current topics by leading experts in the field as well as submitted papers on original and previously unpublished research. Accepted papers will be published in the ARCS Workshop Proceedings as well as in the PARS Newsletter (ISSN 0177-0454). The conference languages are English (preferred) and German. Papers are required to be in English.

A prize of 500 € will be awarded to the best contribution presented personally based on a student's or Ph.D. thesis or project. Co-authors are allowed, the PhD degree should not have been awarded at the time of submission. Candidates apply for the prize by e-mail to the organizers when submitting the contribution. We expect that candidates are or become members of one of the groups ALGO or PARS.

Important Dates
16 March 2020: Deadline for submission of full papers under:
https://easychair.org/conferences/?conf=pasa2020

6 pages in English, formatted according to IEEE CIS template in "conference mode" (see **http://www.ieee.org/conferences_events/conferences/publishing/templates.html**). More details depend on the requirements given by the ARCS organizers and will be provided here as soon as possible

31 March 2020: Notification of authors

13 April 2020: Final version for workshop proceedings

Program Committee
M. Dietzfelbinger (Ilmenau), S. Christgau (Berlin), A. Doering (Zurich), N. Eicker (Jülich), T. Fahringer (Innsbruck), D. Fey (Erlangen), R. Hoffmann (Darmstadt), K. Jansen (Kiel), B. Juurlink (Berlin), ***W. Karl (Karlsruhe)****,* ***J. Keller (Hagen)****, S. Lankes (Aachen), Ch. Lengauer (Passau), E. Maehle (Lübeck), U. Margull (Ingolstadt), E. W. Mayr (Munich), U. Meyer (Frankfurt), F. Meyer auf der Heide (Paderborn), J. Mottok (Regensburg), W. Nagel (Dresden), M. Philippsen (Erlangen), H. Räcke (Munich), K. D. Reinartz (Höchstadt), Ch. Scheideler (Paderborn), B. Schnor (Potsdam), M. Schulz (Munich), U. Schwiegelshohn (Dortmund), P. Sobe (Dresden), C. Trinitis (Munich),* ***R. Wanka (Erlangen)***

Organisation
Prof. Dr. Wolfgang Karl, Karlsruhe Institute of Technology (KIT), Institute of Computer Engineering (ITEC), 76129 Karlsruhe, Phone +49-721-608-43771, E-Mail wolfgang.karl@kit.edu

Prof. Dr. Rolf Wanka, Univ. Erlangen-Nuremberg, Dept. of Computer Science, 91058 Erlangen, Germany, Phone/Fax +49-9131-8525-152/149, E-Mail rolf.wanka@fau.de

ITG

ARCS 2020

33nd GI/ITG INTERNATIONAL CONFERENCE ON ARCHITECTURE OF COMPUTING SYSTEMS
THIS YEAR'S FOCUS: SELF-ORGANIZATION AND SELF-ADAPTION IN HIGH-PERFORMANCE COMPUTING
Aachen, Germany
Mai 25 – 28, 2020
http://arcs2020.itec.kit.edu/

CALL FOR PAPERS

The ARCS conferences series has over 30 years of tradition reporting leading edge research in computer architecture and operating systems. The focus of the 2020 conference will be on concepts and tools for incorporating self-adaptation and self-organisation mechanisms in high-performance computing systems. This includes upcoming approaches for runtime modification at various abstraction levels, ranging from hardware changes to goal changes and their impact on architectures, technologies, and languages.
The proceedings of ARCS 2020 will be published in the Springer Lecture Notes on Computer Science (LNCS) series. A best paper and best presentation award will be presented at the conference.

Paper submission: Authors are invited to submit original, unpublished research papers on one or more of the following topics:

- Hardware Architectures
  - System-on-chip
  - Distributed systems
  - High performance systems
  - Heterogeneous multi- and many-core architectures
  - Architectures for real-time and mixed-criticality systems
  - Coarse- and fine-grained reconfigurable architectures
  - Flexible I/O support
  - Advanced computing architectures
  - New Memory Technologies

- Programming Models and Runtime Environments
  - Programming models for many-core and/or heterogeneous computing platforms
  - Operating systems, hypervisors and middleware for homogeneous and heterogeneous multi-/many-core computing platforms
  - System management including but not limited to scheduling, memory management, power/thermal management, and RTOS

- Cross-sectional Topics
  - Organic Computing
  - Adaptive systems (energy aware, self-x technologies)
  - Pervasive systems
  - Approximate Computing
  - Autonomous systems
  - Support for safety and security

Submission guidelines: Submissions should be done through the link that is provided on the conference website https://easychair.org/conferences/?conf=arcs2020. Papers must be submitted in PDF format.

They should be formatted according to Springer LNCS style (see: https://www.springer.com/gp/computer-science/lncs/conference-proceedings-guidelines) and must not exceed 12 pages, including references and figures.

Workshop and Tutorial Proposals: Proposals for workshops and tutorials within the technical scope of the conference are solicited. Submissions should be done through email directly to the corresponding chair: Carsten Trinitis, (Carsten.Trinitis@tum.de)

Important Dates:

Paper submission deadline (extended):	January 31, 2020
Workshop and tutorial proposals:	TBA
Notification of acceptance:	February 21, 2020
Camera-ready papers:	March 23, 2020

Organizing Committee:

General Chair
Stefan Lankes, RWTH Aachen University, Aachen, Germany
Wolfgang Karl, Karlsruher Institut für Technologie, Karlsruhe, Germany

Program Chairs
Sven Tomforde, Christian-Albrechts-Universität Kiel, Germany
André Brinkmann, Johannes Gutenberg University Mainz, Germany

Workshop and Tutorial Chair
Carsten Trinitis, TU Munich, Germany

Publicity Chair
Lena Oden, Fernuniversität Hagen, Germany

Publication Chair
Thilo Pionteck, Magdeburg University, Germany

Program Committee

Mladen Berekovic, Universität zu Lübeck, Germany
Jürgen Brehm, Leibniz University Hannover, Germany
André Brinkmann, Johannes Gutenberg University Mainz, Germany
Uwe Brinkschulte, University of Frankfurt/Main, Germany
João Cardoso, FEUP/University of Porto, Portugal
Thomas Carle, Institut de Recherche en Informatique de Toulouse, France
Ahmed El-Mahdy, Egypt-Japan University of Science and Technology (E-JUST)
Lukas Esterle, Arrhus University, Danmark

Dietmar Fey, University of Erlangen-Nuremberg, Germany
Giorgis Georgakoudis, Lawrence Livermore National Laboratory, USA
Roberto Giorgi, University of Siena, Italy
Daniel Gracia-Pérez, Thales Research & Technology, France
Jörg Hähner, Augsburg University, Germany
Heiko Hamann, Universität Lübeck, Germany
Andreas Herkersdorf, TU Munich, Germany

Christian Hochberger, TU Darmstadt, Germany
Gert Jervan, Tallinn University of Technology, Estland
Wolfgang Karl, Karlsruher Institut für Technologie, Karlsruhe, Germany
Jörg Keller, Fernuniversität Hagen, Germany
Dirk Koch, University of Manchester, UK
Hana Kubátová, FIT CTU, Prague, Czech Republic
Stefan Lankes, RWTH Aachen University, Germany
Erik Maehle, Universität zu Lübeck, Germany
Lena Oden, Fernuniversität Hagen, Germany
Alex Orailoglu, UC San Diego, USA
Thilo Pionteck, Magdeburg University, Germany
Mario Porrmann, Osnabrück University, Germany
Reza Salkhordeh, Johannes Gutenberg University Mainz, Germany
Toshinori Sato, Fukuoka University, Japan
Martin Schoeberl, University of Denmark, Denmark
Martin Schulz, TU Munich, Germany
Leonel Sousa, University of Lisbon, Portugal
Olaf Spinczyk, Osnabrück University, Germany
Benno Stabernack, Fraunhofer HHI, Germany
Walter Stechele, TU Munich, Germany
Anthony Stein, University of Augsburg, Germany
Jürgen Teich, University of Erlangen-Nuremberg, Germany
Djamshid Tavanagraian, University of Rostock, Germany
Jürgen Teich, University of Erlangen-Nuremberg, Germany
Sven Tomforde, Christian-Albrechts-Universität Kiel, Germany
Carsten Trinitis, TU Munich, Germany
Theo Ungerer, University of Augsburg, Germany
Hans Vandierendonck, Queen's University Belfast, Great Britain
Stephane Vialle, SUPELEC, France
Klaus Waldschmidt, University of Frankfurt, Germany
Dominik Wist, University of Potsdam, Germany
Stephan Wong, Delft University of Technology, The Netherlands

PARS-Beiträge

Studenten	--,-- €
GI-Mitglieder	7,50 €
studentische Nichtmitglieder	-,-- €
Nichtmitglieder	15,00 €

Leitungsgremium von GI/ITG-PARS

Dr. Steffen Christgau, ZIB, Berlin
Dr. Andreas Döring, IBM Zürich
Prof. Dr. Norbert Eicker, FZ Jülich
Prof. Dr. Thomas Fahringer, Univ. Innsbruck
Prof. Dr. Dietmar Fey, Univ. Erlangen
Prof. Dr. Vincent Heuveline, Univ. Heidelberg
Prof. Dr. Ben Juurlink, TU Berlin
Prof. Dr. Wolfgang Karl, Sprecher, KIT
Prof. Dr. Jörg Keller, stellv. Sprecher, FernUniversität in Hagen
Dr. Stefan Lankes, RWTH Aachen
Prof. Dr. Christian Lengauer, Univ. Passau
Prof. Dr.-Ing. Erik Maehle, Universität zu Lübeck
Prof. Dr. Ulrich Margull, TH Ingolstadt
Prof. Dr. Ernst W. Mayr, TU München
Prof. Dr. Jürgen Mottok, OTH Regensburg
Prof. Dr. Wolfgang E. Nagel, TU Dresden
Dr. Karl Dieter Reinartz, Ehrenvorsitzender, Univ. Erlangen-Nürnberg
Prof. Dr. Bettina Schnor, Univ. Potsdam
Prof. Dr. Martin Schulz, TU München
Prof. Dr. Peter Sobe, HTW Dresden
Dr. Carsten Trinitis, TU München
Prof. Dr. Theo Ungerer, Univ. Augsburg
Prof. Dr. Rolf Wanka, Univ. Erlangen-Nürnberg

Sprecher

Prof. Dr. Wolfgang Karl
Karlsruher Institut für Technologie
Institut für Technische Informatik (ITEC)
Haid-und-Neu-Straße 7
76131 Karlsruhe
Tel.: + 49 721 608-43771
Fax: + 49 721 608-43962
E-Mail: wolfgang.karl@kit.edu
URL: http://fg-pars.gi.de/