Performance Evaluation of OSCAR Multi-target Automatic Parallelizing Compiler on Intel, AMD, Arm and RISC-V Multicores

Birk Martin Magnussen[1][0000-0003-2429-9994], Tohma Kawasumi[1], Hiroki Mikami[1], Keiji Kimura[1], and Hironori Kasahara[1]

Department of Computer Science and Engineering, Waseda University
Green Computing Center, 27 Waseda-machi, Shinjuku-ku, Tokyo, 162-0042, Japan
{birk_magnussen, tohma, hiroki}@kasahara.cs.waseda.ac.jp
{keiji, kasahara}@waseda.jp
http://www.kasahara.cs.waseda.ac.jp/index.html.en

Abstract. With an increasing number of shared memory multicore processor architectures, there is a requirement for supporting multiple architectures in automatic parallelizing compilers. The OSCAR (Optimally Scheduled Advanced Multiprocessor) automatic parallelizing compiler is able to parallelize many different sequential programs, such as scientific applications, embedded real-time applications, multimedia applications, and more. OSCAR compiler's features include coarse-grain task parallelization with earliest execution condition analysis, analyzing both data and control dependencies, data locality optimizations over different loop nests with data dependencies, and the ability to generate parallelized code using the OSCAR API 2.1. The OSCAR API 2.1 is compatible with OpenMP for SMP multicores, with additional directives for power control and supporting heterogeneous multicores. This allows for a C or Fortran compiler with OpenMP support to generate parallel machine code for the target multicore. Additionally, using the OSCAR API analyzer allows a sequential-only compiler without OpenMP support to generate machine code for each core separately, which is then linked to one parallel application. Overall, only little configuration changes to the OSCAR compiler are needed to run and optimize OSCAR compiler-generated code on a specific platform. This paper evaluates the performance of OSCAR compiler-generated code on different modern SMP multicore processors, including Intel and AMD x86 processors, an Arm processor, and a RISC-V processor using scientific and multimedia benchmarks in C and Fortran. The results show promising speedups on all platforms, such as a speedup of 7.16 for the swim program of the SPEC2000 benchmarks on an 8-core Intel x86 processor, a speedup of 9.50 for the CG program of the NAS parallel benchmarks on 8 cores of an AMD x86 Processor, a speedup of 3.70 for the BT program of the NAS parallel benchmarks on a 4-core RISC-V processor, and a speedup of 2.64 for the equake program of the SPEC2000 benchmarks on 4 cores of an Arm processor.

Keywords: multicore · parallelizing compiler · OSCAR · multiple platforms · shared memory

1 Introduction

With an increasing number of processor architectures, there is a requirement for supporting multiple architectures in automatic parallelizing compilers.

The OSCAR automatic parallelizing compiler[10] is one such compiler, capable of parallelizing different C and Fortran programs, including scientific applications and simulations, real-time applications, multimedia applications, and more.

Other source-to-source parallelizing compilers have been developed[6, 8] to allow for portability of the generated code between different systems and architectures. The OSCAR compiler is additionally able to output code using the OSCAR API 2.1[15], which is extended from a subset of OpenMP. This allows both OpenMP-capable native compilers to directly compile the OSCAR-compiler generated program, as well as the OSCAR API analyzer to generate separate sequential code for each core of a target system. The resulting sequential code generated by the OSCAR API analyzer for each core then allows a sequential compiler which does not have support for a parallel API such as OpenMP to compile the code for each core and link it to a single parallel program for the target architecture.

Previous evaluations show the performance of OSCAR compiler-generated code on SMP server processors[12], as well as on embedded systems with on-chip shared memory[16].

In this paper, the OSCAR compiler's function, based on multi-grain parallelism and including multiple optimizations such as data localization and cache optimization, will be explained. Additionally, the paper details usage of the OSCAR compiler, targeting systems with and without native compilers supporting OpenMP. Furthermore, this paper analyzes and discusses the performance of programs and benchmarks from the SPEC benchmark suite[9], the NAS parallel benchmark suite[5] and MediaBench II[7], compiled using the OSCAR automatic parallelizing compiler with further optimization techniques such as data localization[22] and cache optimization[13] on different multicore architectures, including an Intel Xeon E5-2650v4 x86 processor, an AMD EPYC 7702P x86 processor, an NVIDIA Carmel ARM®v8.2 processor and a SiFive Freedom U740 RISC-V processor. Neither RISC-V-based processors nor a Zen 2-based processors have been evaluated with the OSCAR compiler before.

2 The OSCAR Automatic Parallelizing Compiler

The OSCAR automatic parallelizing compiler generates parallel code by utilizing multigrain parallelism. Multigrain parallelism includes parallelism of large coarse-grain tasks (coarse-grain parallelism), parallelism of loops (loop-level parallelism) as well as parallelism of individual instructions (statement-level parallelism)[15].

To first exploit coarse grain parallelism, the OSCAR compiler splits the sequential code into macro-tasks. These macro-tasks can be basic blocks of assignments, loops, or function calls. Loops and function calls themselves are then further split into macro-tasks as well. From this, the data and control dependencies between each macro-task can be analyzed, from which, using earliest-execution analysis[10], the macro-tasks are put into a macro-task-graph. The earliest-execution condition for macro-tasks is twofold:

1. A macro-task must wait for the completion of macro-tasks it is directly data-dependent on.
2. A macro-task must wait until preceding control-dependent macro-tasks have evaluated the conditional branches that guarantee said macro-tasks execution, but the macro-task does not need to wait for the completion of these preceding macro-tasks.

Once these two conditions are met, the macro-task can be scheduled into the macro-task-graph. How these conditions are applied in a real program can be seen in Fig. 1. The first condition, that all macro-tasks that the current macro-task to schedule is data dependent on must have finished, can be seen in the `bb19` macro-task. It is scheduled into the macro-task graph once the macro-tasks it is data dependent on, `bb2`, `dosum15`, `bb17`, and `bb18`, are finished. The second condition can be seen for macro-task `bb24`. It can already be scheduled after macro-task `bb5`, since by that time, both its data dependency on `bb1` is fulfilled, and, after the conditional branch `bb5`, it is guaranteed that the control flow will pass `bb24`.

The tasks in the macro-task graph are occasionally shown to have multiple outgoing or incoming dependency edges.

If these edges pass through a dotted arc (representing a logical or), it means that either of the edges passing through the arc will be followed, caused by a conditional branch in the original program. For outgoing edges, it means that only one of the edges will be followed to execute latter macro-tasks, and for incoming edges, it means that only one edge needs to be satisfied to fulfill the dependency and allow execution of the macro-task.

If the edges pass through a solid arc (representing a logical and), it means that all these edges will be followed, caused mostly by coarse-grain task parallelization. For outgoing edges, it means that all these edges will be followed, executing their respective nodes. For incoming edges, it means that all edges must be satisfied to fulfill the dependency and allow execution of the macro-task.

From the macro-task graph, the individual tasks are assigned to the available processor cores. If runtime fluctuation, for example, due to conditional branches, are expected, the OSCAR compiler utilizes dynamic scheduling at runtime to execute the macro-tasks, otherwise, static scheduling is used. The resulting program uses the one time single level thread generation scheme[19], where the program creates a thread per processor core at program start, and the macro-tasks are then run on these threads respectively.

Loop-level parallelism is then applied to doall-loop and reduction-loop type macro-tasks, if possible. Similarly, statement-level parallelism is applied if it is available in a given macro-task[14].

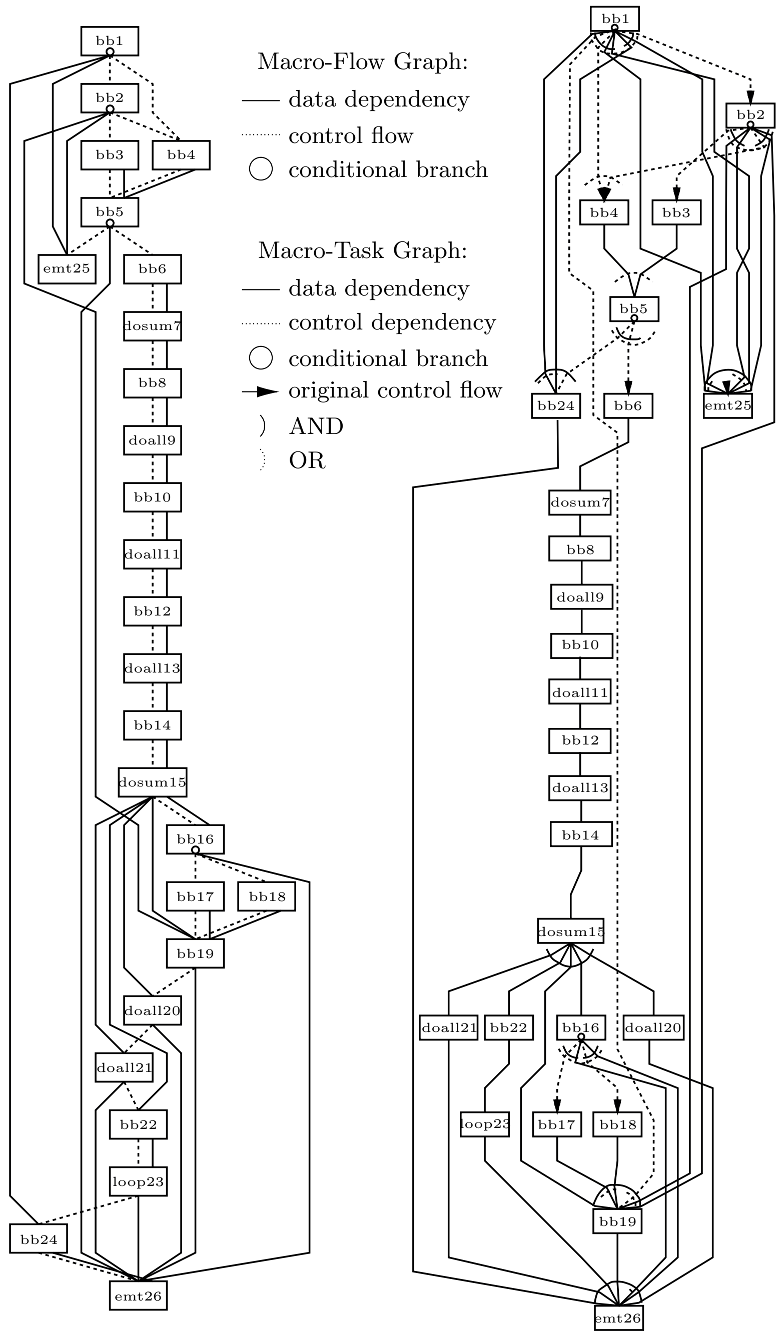

Fig. 1. The macro-flow graph (left) and the macro-task graph (right) of the main training loop of the art benchmark (see section 4)

Additionally, data localization and cache optimization can be performed after macro-task graph generation. Data localization can be performed using loop-aligned decomposition and subsequent generation of data-localization-groups[22]. For this, doall-loop type, reduction-loop type, and sequential-loop type macro-task blocks directly connected only by one data dependence edge are analyzed. In the example macro-task graph in Fig. 1, this would for example be applied to the sequence of macro-tasks from `doall9` to `dosum15`. By calculating which array subscripts in the successive loops are data-dependant on another, the OSCAR compiler can assign sections of the different loops with respective data dependencies into one data-localization-group, which will then be run on one core, in parallel to other data-localization-groups with different sections of the loops of their own. This allows the different data-localization-groups to run in parallel with only minimal data sharing needed at the edge of their data regions. Fig. 2 shows an example of loop-aligned decomposition applied.

Further cache optimization can then be performed by using loop-aligned decomposition, as described above, on loops whose data size exceeds the available cache[13]. With the additional data-localization-groups then potentially exceeding the core count for the target system, executing the groups sequentially will improve the cache behavior of the system. This is because the resulting data-dependent loop sections are small enough to fit their data into the cache, reducing the need to replace the cache while iterating through each loop section. Furthermore, by aligning loop-level parallelism borders to the cache lines, performance can be increased.

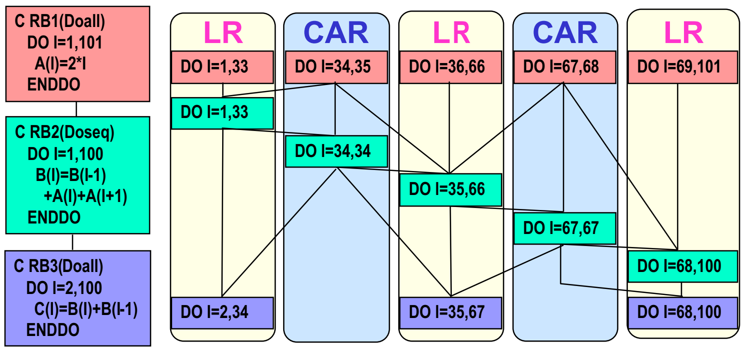

Fig. 2. Example of loop-aligned decomposition of three data-dependent loops. The loops are decomposed into three main localized regions (LR) accessed by one core only, and two commonly accessed regions (CAR) that need to be accessed by multiple cores.

3 Investigated Multicore Architectures

For this paper, four different processor architectures were evaluated. Two x86 processors, one Arm processor, and one RISC-V processor.

The first x86 processor is the Intel Xeon E5-2650v4 12-core processor running at 2.2 GHz with a maximum boost frequency of 2.9 GHz. It has 32 KiB of L1D cache per core, 256 KiB L2 cache per core, and 30 MiB shared L3 cache with a cache line size of 64[11].

The second x86 processor is the AMD EPYC 7702P 64-core processor running at 2 GHz with a maximum boost frequency of 3.35 GHz. It has 32 KiB of L1D cache per core, 512 KiB L2 cache per core, and 256 MiB shared L3 cache grouped into 4-core clusters with a cache line size of 64[3]. If a miss in the L3 cache is available in an L2 cache within the same cluster, the L3 cache can load the data from the L2 cache instead of from the main memory[2].

The Arm processor is the NVIDIA Carmel ARM®v8.2 64 Bit 6-core processor running at 1.4 GHz. It is based on ARMv8.2[4], has 64 KiB L1D cache per core, 2 MiB L2 cache shared between clusters of two cores, and 4 MiB shared L3 cache with a cache line size of 64[18].

The RISC-V processor is the SiFive Freedom U740 4-core SoC running at 1.2 GHz. It has 32 KiB of L1D cache per application core and 2 MiB shared L2 cache with a cache line size of 64[21].

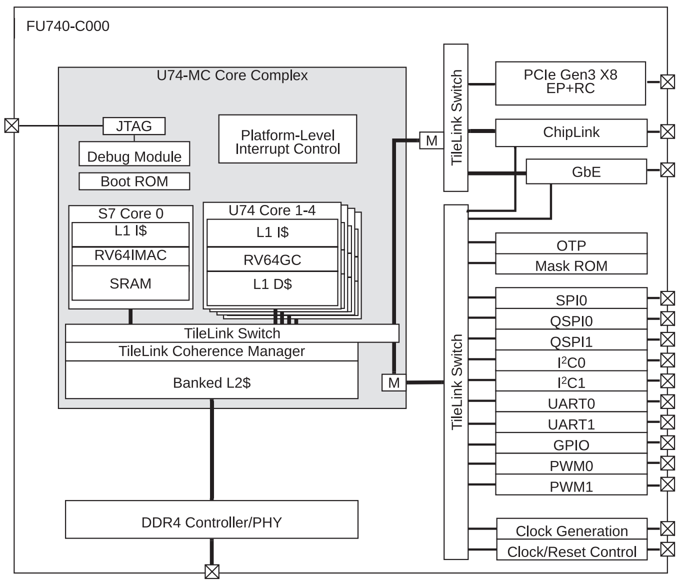

Fig. 3. "FU740-C000 top-level block diagram" by SiFive, Inc.[21]. CC BY-NC-ND 4.0

4 Benchmark Programs

Benchmarks from three different benchmark suites are evaluated in this paper. First, some benchmarks of the NAS parallel benchmark suite[5]. Here, the C version developed by the Real World Computing Project (RWCP), and distributed by the HPCS lab of the University of Tsukuba[20] are used. The specific benchmarks of the NPB suite evaluated are BT, CG, and SP. The SP and BT benchmark both compute the solution of multiple, independent systems of non diagonally dominant equations, an operation used for some computational fluid dynamics algorithms. They differ in the ratio of communication to computation. The CG benchmark applies the conjugate gradient method to a large, sparse, symmetric positive definite matrix to approximate its smallest eigenvalue.

Furthermore, to represent multimedia applications, the MPEG2 encoding benchmark of the MediaBench II suite[7] is evaluated. It is similarly written in C.

Finally, benchmarks of the SPEC2000 floating-point suite[9] are evaluated. From the benchmarks written in C, art and equake are evaluated, and from the benchmarks written in Fortran77, swim is evaluated. The art benchmark test neural network training performance. The equake benchmark simulates seismic wave propagation. The swim benchmark computes a shallow-water model.

The benchmarks in C are manually edited to conform to parallelizable C[17]. Conforming the code to parallelizable C allows the compiler to utilize the full potential of data localization and parallelization. The changes made to the benchmarks are very minor, while some benchmarks do not need any changes at all.

The benchmarks written in Fortran77 are directly passed to the OSCAR compiler with no changes.

5 Compile Flow

First, the target applications are compiled by the OSCAR compiler. For this, the source code must be fed to the respective C or Fortran front-end. The front-end will generate an abstract intermediate representation of the code, which can be analyzed and processed by the middle path of the OSCAR compiler, depending on the necessary optimization options in multiple passes. During this stage, optimization parameters as well as system-dependent optimization information, like the cache architecture of the system, are fed to the OSCAR compiler. Afterward, the resulting optimized intermediate representation file will be fed into the backend, which generates C or Fortran code respectively[12], which is annotated with the OSCAR API 2.1. As the OSCAR API 2.1 is compatible with OpenMP[15], the generated source file can be directly fed into a native compiler, like the GNU C Compiler or the Intel C Compiler, if OpenMP is supported on the target system. If not, because the OSCAR compiler generates synchronization and data transfer code automatically, the OSCAR API analyzer can be utilized to generate code that contains purely sequential code for each CPU core. This allows a sequential-only compiler to generate machine code for each core separately, that can then be linked to a single, parallel application.

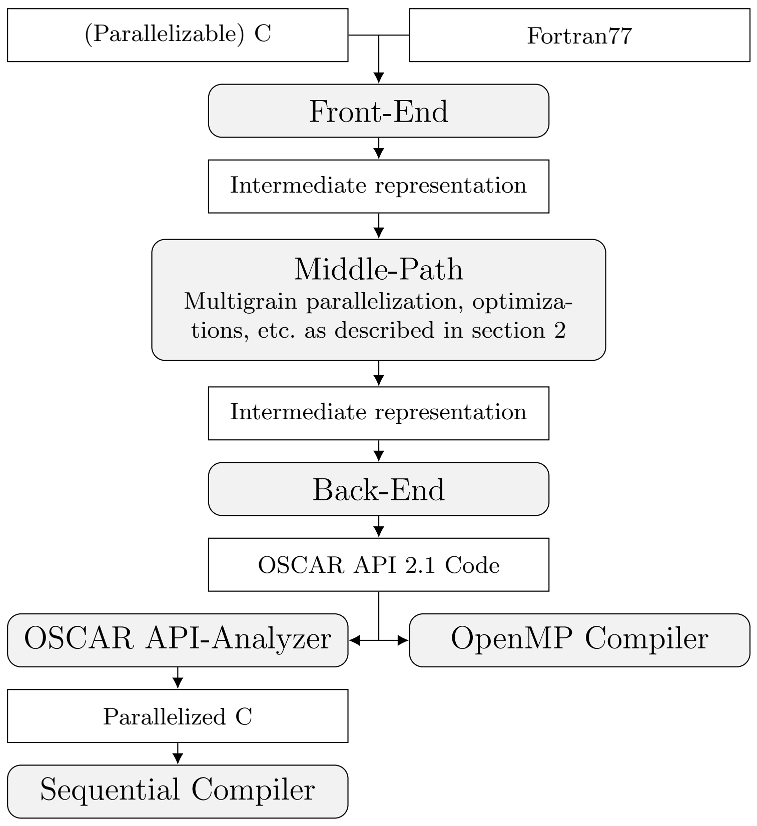

Fig. 4. Compile flow using the OSCAR compiler.

6 Performance of OSCAR Compiler-Parallelized Programs

Unless otherwise noted, all benchmarks, including the sequential reference code and the OSCAR compiler-generated parallelized code, are compiled using the GNU C Compiler with the highest optimization setting (`-Ofast`) and the correct architecture supplied using `-march=`. As all architectures investigated in this paper have native compilers with OpenMP support, the OSCAR compiler-generated code was directly compiled without the use of the OSCAR API-Analyzer. In this paper, the parameters of the OSCAR compiler which are adapted for each target architecture are are focused on cache parameters such as last-level cache size, cache line size, cache associativity, and the number of cores sharing a last-level cache. Other parameters are kept identical across the different architectures. While micro-optimizations with cost tables for the individual architectures are possible, this paper analyzes the performance when using generic cost tables.

6.1 OSCAR Compiled Benchmark Performance on Intel x86

Fig. 5 show the performance of the OSCAR compiler-generated code using a different number of cores, compared to the sequential version of the benchmark on the Intel Xeon E5-2650v4 processor.

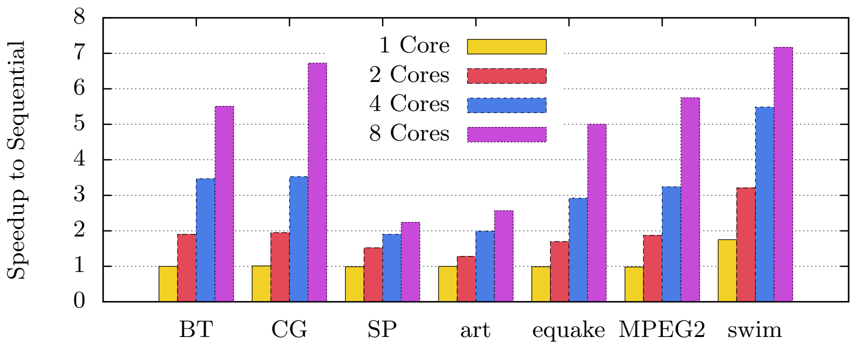

Fig. 5. Relative speedup (higher is better) of the respective benchmark using a certain number of processor cores compared to the sequential version on the Intel x86 processor.

A noticeable result is that the swim benchmark is significantly sped up by using the OSCAR compiler, even with just one core executing the benchmark. At an execution time of 58.1 s for the sequential program and 33.2 s for the OSCAR compiler-generated single-core version, this is a speedup of 1.75. At eight cores, with an execution time of 8.1 s, the speedup is 7.17. The execution times show superlinear speedup for up to 4 cores. This is a result of the cache optimization technique employed by the OSCAR compiler[13] as described in section 2. Table 1 shows the cache statistics of the swim benchmark for the sequential version and the OSCAR compiler-generated versions. These statistics suggest that the OSCAR compiler was able to improve the cache access in the generated code, resulting in the speedup of the benchmark. Furthermore, the MPEG2 encoding benchmark for example, can reduce its execution time from 2.17 s in the sequential version to 0.377 s in the OSCAR compiler-generated eight-core version, for a speedup of 5.75.

To show that the OSCAR compiler can utilize different native compilers, the performances of OSCAR compiler-parallelized benchmarks were tested using the Intel C++ and Fortran Compilers as well. For a better comparison, the sequential reference benchmarks are also compiled using the Intel compilers.

Fig. 6 shows both versions' relative speedup to the sequential execution time of the respective benchmark, run on the Intel Xeon E5-2650v4 processor. This shows that the OSCAR compiler can be used in conjunction with the Intel compilers as well to speed up the final result of the execution. The slightly lower relative speedups compared to Fig. 5 are mostly due to the lower sequential

Table 1. Cache statistics of the swim benchmark as measured by `perf`, sequential version compared to OSCAR compiler-generated version.

Program	L1 loads	L1 load misses	L3 loads	L3 load misses
Sequential	$2.3 \cdot 10^{11}$	$1.2 \cdot 10^{11}$	$5.7 \cdot 10^{10}$	$1.1 \cdot 10^{10}$
OSCAR 1 core	$2.3 \cdot 10^{11}$	$6.5 \cdot 10^{10}$	$1.5 \cdot 10^{10}$	$8.2 \cdot 10^{9}$
OSCAR 2 core	$2.2 \cdot 10^{11}$	$6.5 \cdot 10^{10}$	$1.5 \cdot 10^{10}$	$7.1 \cdot 10^{9}$
OSCAR 4 core	$2.2 \cdot 10^{11}$	$6.5 \cdot 10^{10}$	$1.4 \cdot 10^{10}$	$6.1 \cdot 10^{9}$
OSCAR 8 core	$2.2 \cdot 10^{11}$	$6.5 \cdot 10^{10}$	$1.3 \cdot 10^{10}$	$4.1 \cdot 10^{9}$

execution time of the reference benchmark when compiled with the Intel Compilers at full optimization. For example, while their respective speedups to the sequential versions decreased using the Intel compilers, the absolute execution time of the swim benchmark parallelized using the OSCAR compiler targeting eight cores decreased to 5.8 s, while the execution time of the MPEG2 encoding benchmark decreased to 0.234 s.

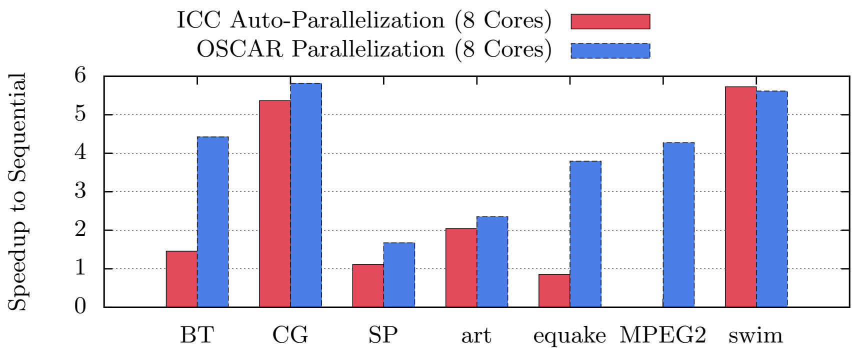

Fig. 6. Relative speedup of the respective benchmark auto-parallelized on 8 cores to its sequential version. Both the sequential version and the OSCAR compiler-generated code used the Intel compiler as the native compiler. The MPEG2 encoding benchmark causes a segmentation fault when compiled with Intel compiler auto-parallelization and run with more than one core, and is thus not shown.

6.2 OSCAR Compiled Benchmark Performance on AMD x86

Fig. 7 show the performance of the OSCAR compiler-generated code using a different number of cores, compared to the sequential version of the benchmark on the AMD EPYC7702P processor.

Similar to the Intel processor, the swim benchmark exhibits high single-core performance and some superlinear speedup due to the cache optimization techniques. Additionally, the CG benchmark shows superlinear speedup. As Fig. 8

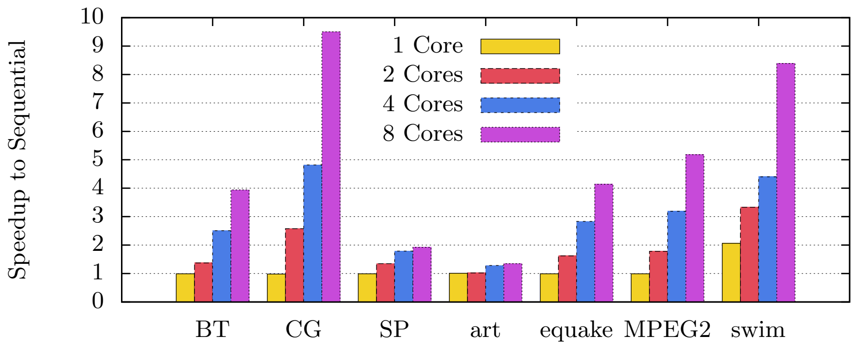

Fig. 7. Relative speedup of the respective benchmark using a certain number of processor cores compared to the sequential version on the AMD x86 processor.

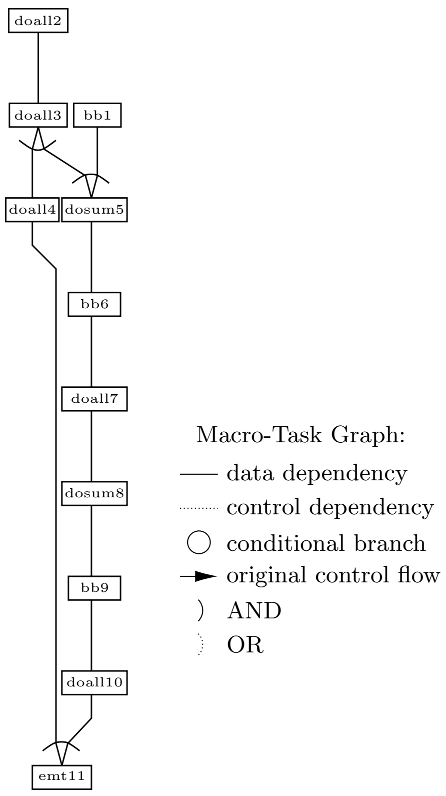

Fig. 8. The macro-task graph of the main loop of the CG benchmark of the NAS parallel benchmark suite (see section 4)

shows, the main loop of CG features many doall loops and dosum reduction loops in succession with data dependence on another. This macro-task graph structure results in data localization similar to the data localization methods using loop-aligned decomposition[22] described in section 2. The data localization is able to improve the performance of the benchmark with multiple cores and causes the superlinear speedup, decreasing overall execution time from 0.86 s in the sequential version to 0.09 s using the OSCAR compiler-generated eight-core version for a 9.5 speedup.

The OSCAR compiler uses operation cost tables for estimating task length for scheduling. These benchmarks used generic tables for all benchmarks. Customizing this table for the AMD EPYC 7702P processor would allow for an improvement of the speedup of the art benchmark on this system. For the art benchmark without customizing the operation cost table, the execution time was only reduced from 4.76 s in the sequential version to 3.52 s using the OSCAR compiler-generated eight-core version for a speedup of 1.35.

6.3 OSCAR Compiled Benchmark Performance on Arm

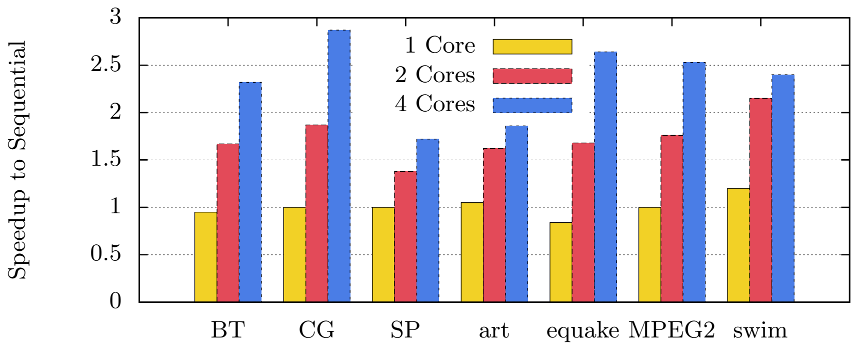

Fig. 9. Relative speedup of the respective benchmark using a certain number of processor cores compared to the sequential version on the Arm processor.

Fig. 9 shows the performance of the OSCAR compiler-generated code using a different number of cores, compared to the sequential version of the benchmark on the NVIDIA Carmel ARM®v8.2 64 Bit processor.

The Arm processor shows overall good speedup for the different benchmarks. While the cache optimization applied to the swim benchmark is noticeable, it is much smaller compared to the effects on the Intel and AMD CPU's. Overall, good speedup is observed, with for example the equake benchmark's execution time decreasing from 19.0 s in the sequential version to 7.18 s using the OSCAR compiler-generated four-core version for a speedup of 2.64.

6.4 OSCAR Compiled Benchmark Performance on RISC-V

Fig. 10 show the performance of the OSCAR compiler-generated code using a different number of cores, compared to the sequential version of the benchmark on the SiFive Freedom U740 processor.

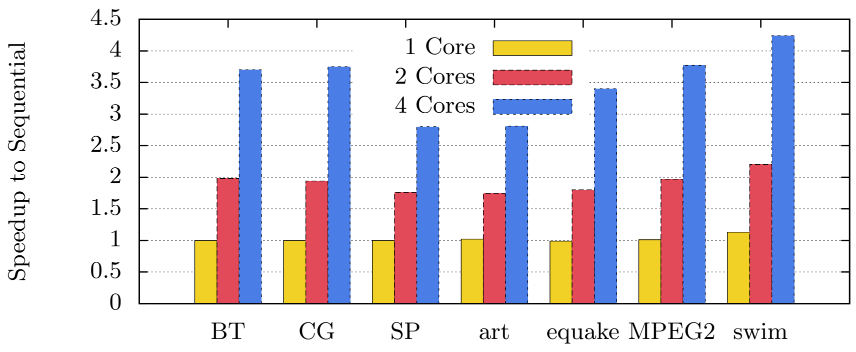

Fig. 10. Relative speedup of the respective benchmark using a certain number of processor cores compared to the sequential version on the RISC-V processor.

The RISC-V processor shows good overall speedup as well. Notably, the observed speedup is much more homogeneous compared to the other platforms. This is because the RISC-V SoC is comparably slower than the other processors used, while the memory performance is similar to the other systems. This reduces the overall effect of memory on the benchmarks, both the bottlenecks, as well as the positive impact of cache optimization for benchmarks like swim. For example the BT benchmark's execution time decreased from 2041 s in the sequential version to 551 s using the OSCAR compiler-generated four-core version for a speedup of 3.7.

7 Conclusion

This paper shows how the OSCAR automatic parallelizing compiler can utilize multigrain parallelism to generate parallelized code from different programs and benchmarks for various architectures from embedded multicores to high-performance processors. This code can then be compiled for the respective architectures using a native compiler of the system, like the Intel C and Fortran Compilers or the GNU C and Fortran Compilers. By utilizing the OSCAR API 2.1 and the OSCAR API analyzer, it is even possible to generate sequential code for each core of a system, which allows OSCAR compiler-generated code to be run on systems whose native compilers do not support a parallel API such as OpenMP. Overall, this paper finds that the OSCAR compiler is able to automatically parallelize a variety of benchmarks, including scientific simulations, media applications, and machine learning applications, written in C and Fortran with good speedup. Measuring the benchmark programs showed good performances, such as a speedup of 7.16 for the swim program of the SPEC2000 benchmarks, a speedup of 6.73 for the CG program of the NAS parallel benchmarks, and a speedup of 5.75 for the MPEG2 encoding benchmark on 8 cores of an Intel x86 processor. Similarly, on 8 cores of the AMD x86 processor, speedups such as 9.50 for the CG program of the NAS parallel benchmarks, a speedup of 8.39 for the swim program of the SPEC2000 benchmarks, and a speedup of 3.94 on the BT benchmark of the NAS parallel were observed. On the 4-core SiFive RISC-V processor, speedups including a speedup of 3.70 for the BT program of the NAS parallel benchmarks on a 4-core, a speedup of 2.80 for the SP program of the NAS parallel benchmarks, and a speedup of 3.40 for the equake program of the SPEC2000 benchmarks. Finally, on 4 cores of an NVIDIA Arm processor, observed speedups include 2.64 for the equake program of the SPEC2000 benchmarks, 2.87 for the CG program of the NAS parallel benchmarks, and 1.86 for the art program of the SPEC2000 benchmarks.

These speedups are similar to the previous performance of OSCAR generated code on the RP2 processor platform[1]. This shows that the OSCAR compiler can achieve good speedup and performance for benchmarks on different architectures with different instruction sets as well. The OSCAR compiler proves to be able to handle parallelizing code for a variety of current architectures, including embedded systems and high-performance processors, without extensive per-system tuning, using default parameters and cost tables.

Due to advanced optimization techniques such as cache optimization and data localization, superlinear speedup can be achieved for some benchmarks.

References

1. Adhi, B.A., Kashimata, T., Takahashi, K., Kimura, K., Kasahara, H.: Compiler software coherent control for embedded high performance multicore. IEICE Transactions on Electronics **E103.C**(3), 85–97 (2020). https://doi.org/10.1587/transele.2019LHP0008
2. Advanced Micro Devices, Inc.: Software Optimization Guide for AMD Family 17h Processors (2017)
3. Advanced Micro Devices, Inc.: Preliminary Processor Programming Reference (PPR) for AMD Family 17h Model 31h, Revision B0 Processors (2020)
4. Arm Limited: Arm® Architecture Reference Manual, Armv8, for Armv8-A architecture profile (2021)
5. 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., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K.: The NAS parallel benchmarks summary and preliminary results. In: Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing. pp. 158–165 (1991). https://doi.org/10.1145/125826.125925
6. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T.: Parallel programming with polaris. Computer **29**(12), 78–82 (1996). https://doi.org/10.1109/2.546612

7. Fritts, J.E., Steiling, F.W., Tucek, J.A., Wolf, W.: MediaBench II video: Expediting the next generation of video systems research. Microprocessors and Microsystems **33**(4), 301–318 (2009). https://doi.org/10.1016/j.micpro.2009.02.010
8. Hall, M., Anderson, J., Amarasinghe, S., Murphy, B., Liao, S.W., Bugnion, E., Lam, M.: Maximizing multiprocessor performance with the SUIF compiler. Computer **29**(12), 84–89 (1996). https://doi.org/10.1109/2.546613
9. Henning, J.L.: SPEC CPU2000: Measuring CPU performance in the new millennium. Computer **33**(7), 28–35 (Jul 2000). https://doi.org/10.1109/2.869367
10. Honda, H., Kasahara, H.: Coarse grain parallelism detection scheme of a fortran program. Systems and Computers in Japan **22**(12), 24–36 (1991). https://doi.org/10.1002/scj.4690221203
11. Intel Corp.: Intel® 64 and IA-32 Architectures Software Developer's Manual (2021)
12. Ishizaka, K., Miyamoto, T., Shirako, J., Obata, M., Kimura, K., Kasahara, H.: Performance of OSCAR Multigrain Parallelizing Compiler on SMP Servers. In: Languages and Compilers for High Performance Computing. pp. 319–331. Springer Berlin Heidelberg (2005). https://doi.org/10.1007/11532378_23
13. Ishizaka, K., Obata, M., Kasahara, H.: Coarse grain task parallel processing with cache optimization on shared memory multiprocessor. In: Proceedings of the 14th International Conference on Languages and Compilers for Parallel Computing. p. 352–365. LCPC'01, Springer-Verlag, Berlin, Heidelberg (2001). https://doi.org/10.1007/3-540-35767-X_23
14. Kimura, K., Wada, Y., Nakano, H., Kodaka, T., Shirako, J., Ishizaka, K., Kasahara, H.: Multigrain parallel processing on compiler cooperative chip multiprocessor. In: 9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05). pp. 11–20 (2005). https://doi.org/10.1109/INTERACT.2005.9
15. Kimura, K., González-Alvarez, C., Hayashi, A., Mikami, H., Shimaoka, M., Shirako, J., Kasahara, H.: OSCAR API v2.1: Extensions for an advanced accelerator control scheme to a low-power multicore API. In: 17th Workshop on Compilers for Parallel Computing (2013)
16. Kimura, K., Mase, M., Mikami, H., Miyamoto, T., Shirako, J., Kasahara, H.: OSCAR API for real-time low-power multicores and its performance on multicores and SMP servers. In: Languages and Compilers for Parallel Computing. pp. 188–202. Springer Berlin Heidelberg (2010). https://doi.org/10.1007/978-3-642-13374-9_13
17. Mase, M., Onozaki, Y., Kimura, K., Kasahara, H.: Parallelizable c and its performance on low power high performance multicore processors (2010)
18. NVIDIA Corp.: NVIDIA Jetson Xavier NX System-on-Module Data Sheet (2020)
19. Obata, M., Shirako, J., Kaminaga, H., Ishizaka, K., Kasahara, H.: Hierarchical parallelism control for multigrain parallel processing. In: Languages and Compilers for Parallel Computing. pp. 31–44. Springer Berlin Heidelberg (2005). https://doi.org/10.1007/11596110_3
20. Real world computing project: Omni OpenMP Compiler Project. http://www.hpcs.cs.tsukuba.ac.jp/omni-compiler/, accessed: 2021-07-18
21. SiFive, Inc.: SiFive FU740-C000 Manual (2021)
22. Yoshida, A., Koshizuka, K., Kasahara, H.: Data-localization for fortran macro-dataflow computation using partial static task assignment. In: Proceedings of the 10th International Conference on Supercomputing. p. 61–68. ICS '96, Association for Computing Machinery, New York, NY, USA (1996). https://doi.org/10.1145/237578.237586