Vol.2012-ARC-202 No.10
Vol.2012-HPC-137 No.10
2012/12/13

Automatic parallelization with OSCAR API Analyzer: a cross-platform performance evaluation

Cecilia Gonzalez-Alvarez[1,2,a)] Youhei Kanehagi[1,b)] Kosei Takemoto[1,c)] Yohei Kishimoto[1,d)]
Kohei Muto[1,e)] Hiroki Mikami[1,f)] Akihiro Hayashi[1,g)] Keiji Kimura[1,h)]
Hironori Kasahara[1,i)]

Abstract: To satisfy the demands of auto parallelizing compilers in the diverse industry of multicores, we have developed the OSCAR API Analyzer. It allows programs automatically parallelized by the OSCAR compiler with OSCAR API directives to target many different platforms using just sequential compilers. We have evaluated the execution performance of the parallelization of Fortran SPEC benchmarks (tomcatv, swim2000, mgrid2000) and media C benchmarks (AAC encoder, Optical flow, MPEG2 encoder, MPEG2 decoder, Face detect) on five HPC servers and four embedded multicores. Speedups on servers were up to 18x for 32 cores (swim2000 on Hitachi SR16000), whereas on embedded systems, AAC encoder speedup was up to 47x on TilePro64, for 64 homogeneous cores, and up to 32.65x for the optical flow on the heterogeneous multicore RP-X, using 8 cores and 4 accelerators.

1. Introduction and Motivation

In the latest years, multicores have invaded every spot of the computing spectrum, from High Performance Computing (HPC) to embedded systems. Specifically, the recent rise and expected continuous grow of embedded multicores in new markets like smartphones or advanced automobile control suggests that research directions will be focused on leveraging high-performance and low power multicore embedded systems. One of the top priorities of embedded companies is the portability of applications and libraries. Therefore, automatic parallelizing compilers that can target multiple multicore platforms are of great interest for the current market. If embedded companies can minimize their investment in compiler development, they will get parallel code quickly to satisfy the tightly product release plan, as it is the case in the Japanese smartphone market every Spring and Fall.

There have been several source-to-source compilers that perform automatic parallelization, such as SUIF [7] or Polaris [4]. Lately, there parallel APIs are also becoming of special interest, like The Multicore Association communication API [21] for embedded systems, or OpenCL [11], mostly used on systems with accelerators like GPGPUs.

1 Waseda University
2 Universitat Politecnica de Catalunya
a) cecilia@kasahara.cs.waseda.ac.jp
b) ykane@kasahara.cs.waseda.ac.jp
c) kosei@kasahara.cs.waseda.ac.jp
d) kisimoto@kasahara.cs.waseda.ac.jp
e) kmuto@kasahara.cs.waseda.ac.jp
f) hiroki@kasahara.cs.waseda.ac.jp
g) ahayashi@kasahara.cs.waseda.ac.jp
h) kimura@apal.cs.waseda.ac.jp
i) kasahara@kasahara.cs.waseda.ac.jp

Several compilers that generate parallel code for shared-memory multiprocessors use the de-facto standard multicore API OpenMP [17]. However, compilers for each existing system do not always support OpenMP, as it is the case of embedded systems.

In previous works, Kasahara *et al.* presented OSCAR (Optimally SCheduled Advanced multiprocessoR) compiler [9], a source-to-source compiler that extracts multigrain parallelism of C and Fortran applications. The compiler rewrites the programs with OSCAR API [18], also presented by them in a previous work [13]. OSCAR API includes a subset of OpenMP for thread control and multiprocessor task management, such as memory allocation, power control, data transfer by DMAC, realtime execution, accelerator control and cache control. Still, the specific part of OSCAR API that can be applicable depends on the target platform. For instance, parallel processing on SMP servers uses only the parallel thread generation and memory order management directives of the API, whereas certain heterogeneous systems can use as well the memory map API, the power API and the accelerator API. Furthermore, run-time implementations of the API may differ across the platforms.

In this paper, we extend the above mentioned works with the following contributions:

- A new compilation flow for automatic parallelization of C and Fortran applications that abstracts the target run-time with an API Analyzer. It enables the rapid development of parallelizing compilers for homogeneous and heterogeneous multicores, using each platform sequential compiler as the backend compiler. Consequently, platforms that do not support OpenMP compilers can also benefit from automatic parallelization.

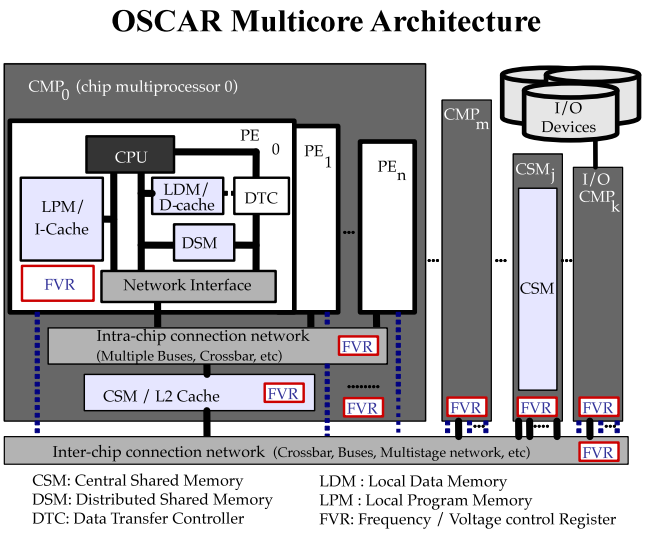

Fig. 1 **OSCAR multicore architecture diagram.** Each core has a CPU with instruction- and data caches (I/D-cache), local data memory (LDM), L2-cache and distributed shared memory (DSM). Peripherals are the DMA-controller (DTC) and a network interface to the on-chip network. Components can have frequency-voltage registers (FVR). External memories (CSM) are accessed by the inter-chip connection network. Concrete implementation of the architecture is abstracted.

- Evaluation of performance and scalability of the proposed automatic parallelizing framework in several shared-memory HPC systems, several homogeneous multicore embedded systems and a heterogeneous multicore system. As a proof of the validity of the new framework compilation with sequential compilers as the backend, we compare the results against executables generated with OpenMP backend compilers.

The key components of the paper are the **multiplatform parallelizing compilation flow** with OSCAR compiler as the automatic parallelization compiler, and **OSCAR API Analyzer** that enables parallelization of several multiprocessors servers and embedded multicores with almost no development costs.

The rest of the paper is organized as follows. Section 2 explains OSCAR compiler and OSCAR API. Section 3 explains OSCAR API Analyzer implementation details. In Section 4, we present the evaluation results of the automatic parallelization of benchmarks on HPC and embedded systems. Section 5 explores the related work and Section 6 finalizes with the conclusions.

2. OSCAR infrastructure

OSCAR Multicore architecture [12] is an architecture abstraction designed to work cooperatively with a multigrain parallelizing compiler such as OSCAR compiler [9]. The goal of OSCAR Multicore is to build a scalable, highly performance and cost effective computer system for various targets, from embedded computers like the ones found on smartphones, automobiles, PDAs, game machines and medical systems, to HPC systems.

The OSCAR reference architecture is shown in Figure 1. It consists of several multicore chips and an off-chip Centralized Shared Memory (CSM) module. Each multicore chip has multiple processor cores and an on-chip CSM. Each processor core has a CPU, a Local DataMemory and Local ProgramMemory (LDM, LPM) for core private data and instructions, a Distributed Shared Memory (DSM) for synchronization flags and shared data, a Data Transfer Controller (DTC), a Frequency and Voltage Control Register (FVR) and a network interface that connects Processor Elements (PE) within a chip. Each module in the OSCAR memory architecture may have an FVR. The OSCAR compiler and API can support several memory configurations and multicores that are subsets of the reference architecture designed for each target application.

OSCAR compiler exploits multiple grains of parallelism such as coarse-grain task parallel processing, loop-iteration level parallel processing and statement level near fine-grain parallel processing [12]. It consists of the following phases: front end, that performs lexical and syntax analysis and generates an intermediate representation (IR); middle path, that performs macro-task generation [10], coarse-grain task parallelization, processor grouping and macro-task scheduling, data locality optimization and power reduction; back end, that generates binary or source code with OSCAR API.

OSCAR API is an interface that provides parallel execution, memory allocation, data transfer by DMAC, power control, synchronization, realtime execution, cache control and accelerator control. The OSCAR API is designed on a subset of OpenMP. Therefore, a program that is automatically parallelized by the OSCAR compiler with OSCAR API can be compiled by OpenMP compilers and can support both C and Fortran programs. The directives in the OSCAR API are related to the above modules of the OSCAR Multicore architecture. If the target architecture does not have some of those modules, directives related to those modules can be ignored.

Figure 2 shows the compilation flow of the OSCAR compiler with the OSCAR API. In the first step, a sequential Parallelizable C or Fortran program is parallelized by the OSCAR compiler. Parallelizable C is a set of coding guidelines for C programs taking compilers parallelization in consideration [16]. For heterogeneous architecture, the accelerator compiler or the programmer first inserts hint directives to indicate OSCAR compiler the parts of the code that can be offloaded to accelerators [8]. The generated C or Fortran code includes OSCAR API directives. In the last step the OSCAR API Analyzer transforms the directives into function calls to the OSCAR run-time support. Since the directives are a subset of OpenMP it is also possible to feed the sources into an OpenMP-compliant compiler. However, the API Analyzer is easier to port to new platforms than modifying OpenMP-compliant compilers such as GCC, as experiments in our group have shown. Futhermore, the OSCAR API Analyzer is able of processing accelerator directives.

3. OSCAR API Analyzer

The OSCAR API Analyzer enables the use of the memory mapping API and the power control API of OSCAR API in any given target platform. The tool interprets parts of platform dependent directive statements and transforms program sources to use standard library calls. The sources that are generated from this

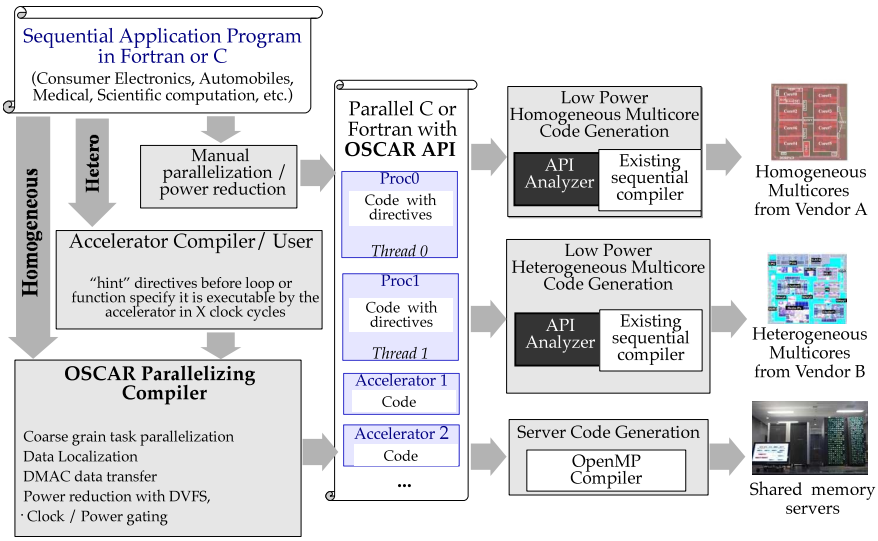

Fig. 2 OSCAR compilation flow for homogeneous and heterogeneous multicore systems.

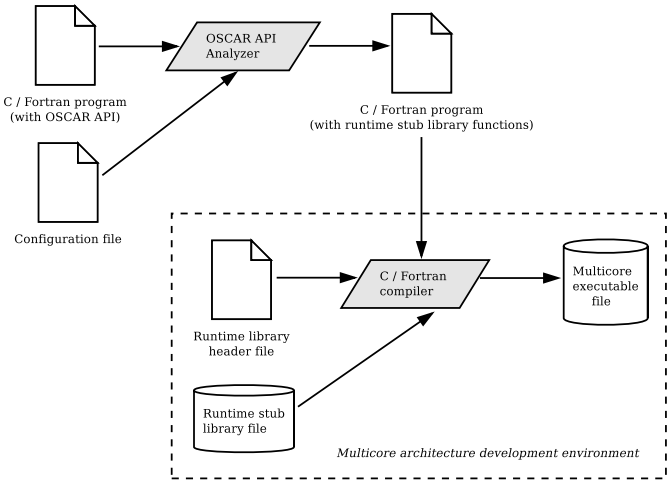

Fig. 3 Diagram of the OSCAR API Analyzer workflow.
The OSCAR API Analyzer reads in a configuration file and the C/Fortran sources compiled by tset he OSCAR parallelizing compiler that now contain *directives*. The Analyzer transforms the directives into run-time library calls for the target platform, that sequential compilers are able to handle to generate the final multicore-enabled binary.

transformation are linked with run-time libraries that are easy to create. OSCAR compiler can be used to parallelize programs automatically and to perform power control with almost no development efforts. Therefore, the development cost of using the OSCAR API Analyzer is considerably lower than developing a parallelizing compiler for every target architecture, enabling automatic parallelization in a wide range of different architectures.

3.1 OSCAR API Analyzer Workflow

Figure 3 shows the workflow of the OSCAR API Analyzer. Inputs of the workflow are C or Fortran code files that have been automatically parallelized with OSCAR compiler and thus use directives of the OSCAR API. To create an executable for each one of the target parallel architectures we compile the generated C or Fortran program with run-time library functions and link with the specific run-time library of the architecture.

Outputs of the OSCAR API Analyzer are C or Fortran parallelized programs that use run-time library functions. The configuration file that the Analyzer uses for each platform describes and sets the architecture configuration, the address of the distributed shared memory (DSM), transformations of the data transfer, `groupbarrier` directive to implement barrier synchronization among any group of processor cores, the associated module name and module number for the power control API.

3.2 Runtime Library Functions Transformation

The core task of OSCAR API Analyzer is the transformation of OSCAR API into run-time library functions. Taking the example of a C program that has been parallelized with OSCAR compiler, the OSCAR API directives are specified as pragmas in the C code. Therefore, to be able to use the functions of a standard parallel library, the OSCAR API Analyzer has to transform the OSCAR API directives into run-time stub library functions that call functions of the standard parallel library of the target.

Figure 4 shows an example of conversion of parallel sections to perform thread creation. In Figure 4(a) 4 threads are created using the `parallel sections` directive. The OSCAR API Analyzer generates the code in Figure 4(b). Functions `thread_function_00[0-3]` start the execution of the threads, but while `function thread_function_000` runs as a normal function, `oscar_thread_create` prepare the execution of functions `thread_function_00[1-3]`. Finally, `oscar_thread_join` function performs the synchronization of the threads.

In platforms where the pthread library is available, functions `oscar_thread_create` and `oscar_thread_join` call `pthread_create` and `pthread_join`, respectively.

```
OMP_NUM_THREADS=4

#pragma omp parallel_sections
{
  #pragma omp section {
    . . .
  }
  #pragma omp section {
    . . .
  }
  #pragma omp section {
    . . .
  }
  #pragma omp section {
    . . .
  }
}
```

(a) OSCAR API directives

```
void thread_function_000(void);
void thread_function_001(void);
void thread_function_002(void);
void thread_function_003(void);

int thr1;
int thr2;
int thr3;

oscar_thread_create( &thr1, thread_function_001, 0);
oscar_thread_create( &thr2, thread_function_002, 1);
oscar_thread_create( &thr3, thread_function_003, 2);

thread_function_000();
oscar_thread_join(thr1);
oscar_thread_join(thr2);
oscar_thread_join(thr3);
```

(b) Runtime library functions

Fig. 4 OSCAR API Analyzer conversion example.
Figure (a) shows the directives generated by OSCAR compiler. Each `omp section` is mapped to a thread. Figure (b) shows the output of the OSCAR API Analyzer. The *directives* have been translated to run-time library calls.

```
#pragma oscar onchipshared(var1)
```

(a) OSCAR API directive

```
int __attribute((section("OSCAR_SHARED")))var1;
```

(b) code for GCC conversion

Fig. 5 OSCAR API-Analyzer conversion example.
The `onchipshared` directive is transformed into a GCC-specific compiler annotation for the variable `var1`.

3.3 Memory Allocation Transformation

The run-time library cannot implement the memory allocation as it is stated in directives such as `thread_private`. Therefore, it is necessary to convert the memory allocation specification of OSCAR API to a description of the memory attribute layout used by the native compiler.

Figure 5 shows an example of this tranformation for the case of the directive `onchipshared`. Figure 5(a) displays how `onchipshared` directive allocates `var1` in the on-chip centralized shared memory (CSM). Figure 5(b) shows the transformation when GCC is the native compiler: the variable `var1` is declared as an attribute in the section `OSCAR_SHARED`. As the linker configuration file of the target platform places the `OSCAR_SHARED` section in the on-chip CSM, variable var1 will consequently be allocated there.

Figure 6 shows another example of OSCAR API transformation related to memory allocation, in this case for the directives `threadprivate` and `distributedshared`, under the assumption of using SH-compiler as the native compiler. These directives specify that variables `var1` and `var2` are allocated in the on-chip shared memory (CSM) and the distributed shared memory, respectively (Figure 6(a)). The OSCAR API Analyzer transforms those directives into the code shown in Figure 6(b), with sections `MEM_LM` and `MEM_DSM` describing that variables `var1` and `var2` are placed in Local Data Memory (LDM) and Distributed Shared Memory (DSM).

```
int  var1[10];
short int var2 [10][10];
#pragma omp threadprivate(var1)
#pragma oscar distributedshared vpc(0) (var2)
```

(a) OSCAR API directives

```
#pragma section MEM_LM
int  var1[10];
#pragma section
#pragma section MEM_DSM0
short int  var2[10][10];
```

(b) Output code of OSCAR API Analyzer

Fig. 6 OSCAR API-Analyzer conversion example.
In Figure (a) two variables (`var1` and `var2`) are mapped to thread private and distributed shared memory. After translation with the OSCAR API Analyzer - see Figure (b), the variables are mapped to the platform specific memory areas: `MEM_LM` and `MEM_DSM0`.

4. Evaluation results

We evaluate the performance and scalability of several benchmarks on HPC servers and embedded systems. We also compare the results to their OpenMP counterpart when applicable.

The benchmarks consist of Fortran and C codes for the servers, and only C codes for the embedded machines. Note that some of the original C codes of the applications had been modified to made them compliant with the Parallelizable C language [16].

The figures show the number of parallel cores on the x-axis and the application speedup on the y-axis. Each bar series represents a benchmark compiled with the OSCAR API Analyzer and a sequential compiler. If an OpenMP compiler is available for the evaluated platform, bar series also represent executables generated by OpenMP compilers. In that case, `(omp)` or `(api)` follows the name of the benchmark if an OpenMP compiler or the OSCAR API Analyzer have been used, respectively.

4.1 HPC systems

Experimental setup

Five different HPC servers were used to run the experiments: Fujitsu m9000 (256-core SMP), Hitachi SR16000 (128-core SMP), Hitachi SMP 64-core Blade server BS2000, Hitachi Intel-core SMP RS440 and Dell PowerEdge AMD-core SMP R815. Table 4.1 shows the configuration details of these servers.

The sequential compiler used to ultimately generate the executables is GCC. All the applications have been compiled with optimization level flag `-O3`.

Fortran benchmarks

We test the compilation flow with Fortran applications from the SPEC Benchmarks [20]; they appear in the charts of this paper as `tomcatv`, `swim2000` and `mgrid2000`. We test them on Hitachi Power7 SR16000 and Hitachi Intel-core SMP RS440 servers.

Table 1 Evaluation environment (HPC systems)

System	Fujitsu m9000	Hitachi SR16000	Hitachi BS2000	Hitachi RS440	Dell Power Edge R815
CPU	SPARC64 VII 2.88GHz	IBM POWER 7 4GHz	Intel Xeon E7-8870 2.4GHz	Intel Xeon X7560 2.27GHz	AMD Opteron 6174 2.20MHz
#cores	256	128	80	32	48
L1 Cache	128KB / core	32KB / core	48KB / core	32KB (I); 16KB (D)	128KB / core
L2 cache	6MB / CPU	256KB / core	256KB / core	256KB / core	512KB / core
L3 cache	N/A	32MB / 8 cores	30MB / CPU	24MB / CPU	12MB / 12 cores

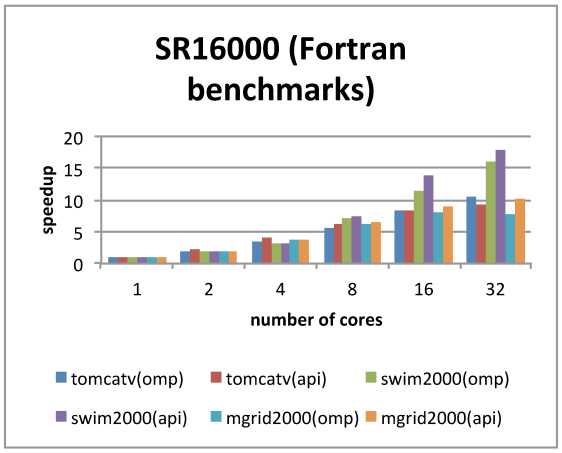

Fig. 7 Performance of Fortran benchmarks on Hitachi Intel-core SMP RS440.

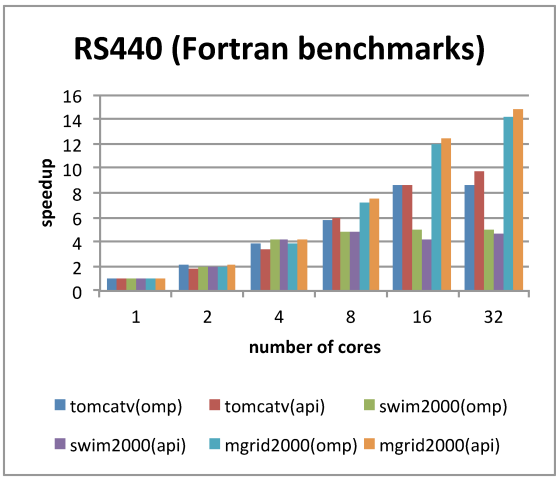

Fig. 8 Performance of Fortran benchmarks on Hitachi Intel-core SMP RS440.

Figure 7 shows the results for the SR16000 server. The benchmarks with OSCAR API Analyzer support can reach a speedup up to 18x, as it is the case of `swim2000` for 32 cores (*swim2000(api)*-labeled bar). Besides, we get up to 2.5x more speedup with `mgrid2000` (*mgrid2000(api)*-labeled bar) compared to the OpenMP execution (*mgrid2000(omp)*-labeled bar). The reason of this speedup difference between the OpenMP version and the Analyzer version is due to a more efficient implementation of the run-time libraries that the Analyzer uses, compared to the OpenMP run-time for this particular system.

Figure 8 shows the results for the RS440 server. In this case, OSCAR API Analyzer gives also the best performance, up to 14.85x of speedup for `mgrid2000` with 32 cores (*mgrid2000(api)*-labeled bar). This means that using OSCAR API with the API Analyzer and sequential compilers as backend compilers give a similar or better performance than the OpenMP compilers as the backend compilers.

C benchmarks

The C benchmarks used in the evaluation of the servers are: AAC encoder (`aac`), Optical Flow (`optflow`), MPEG2 encoder (`m2enc`) and Face detect (`fd`). These benchmarks are from known benchmark suites such as Mediabench and OpenCV, or obtained from collaboration with other research groups. They were rewritten in Parallelizable C, except for `aac`, that was originally written in Parallelizable C. We test their performance on Fujitsu SPARC-VII m9000, Hitachi Power7 SR16000, Hitachi SMP Blade server BS2000, Hitachi Intel-core SMP RS440 and Dell PowerEdge AMD-core SMP R815.

Figure 9 shows the performance results on m9000. This machine has no results for `m2enc` because of unsupported libraries in the platform. The maximum performance achieved is 10x of speedup with the `fd` benchmark parallelized for 16 cores with the

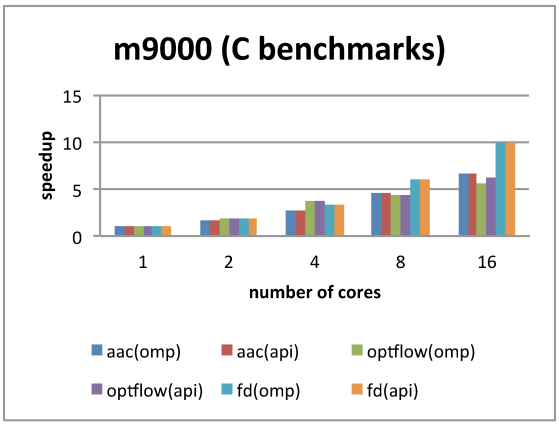

Fig. 9 Performance of C benchmarks on Fujitsu SPARC-VII m9000.

OSCAR API Analyzer (*fd(api)*-labeled bar).

The server SR16000 has a similar performance, as we see in Figure 10, reaching 19.15x of speedup also with the `fd` benchmark and 32 cores (*fd(api)*-labeled bar). However, scalability for `m2enc` degrades from 16 cores (*m2enc(api)*-labeled bar).

In turn, the scalability of all the benchmarks on BS2000 (Figure 11) grows more steadily. `aac` gives the best result with 11.8x for the Analyzer version with 16 cores (*aac(api)*-labeled bar).

Results for the RS440 machine, shown in Figure 12, give the best speedup also to `aac`, 19x for 32 cores (*aac(omp)*-labeled bar), but followed closely by the Analyzer with 18.85x (*aac(api)*-labeled bar). However, in this machine, the scalability of `optflow` and `m2enc` from 16 cores does not improve for either OpenMP or the Analyzer.

We see in Figure 13, that shows the results for R815, a common trend for the execution of the C applications on the servers: all the

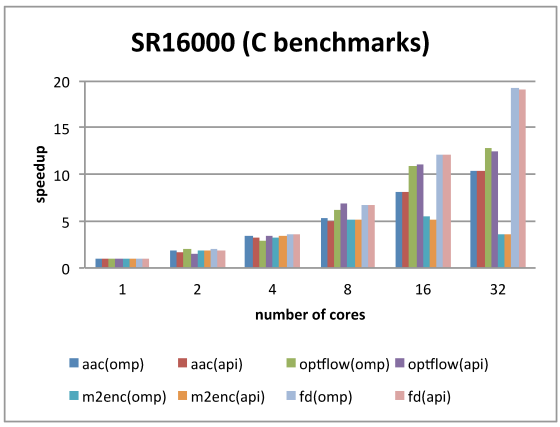

Fig. 10 Performance of C benchmarks on Hitachi Power7 SR16000.

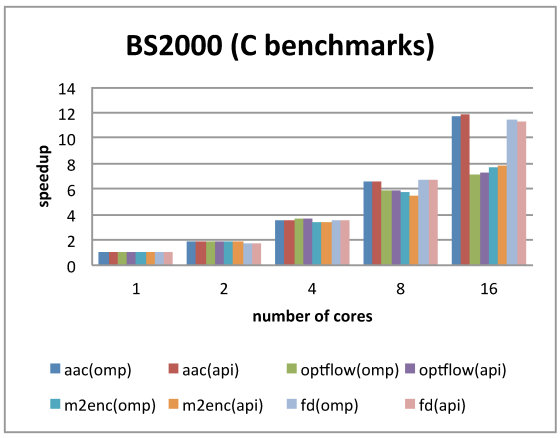

Fig. 11 Performance of C benchmarks on Hitachi SMP Blade server BS2000.

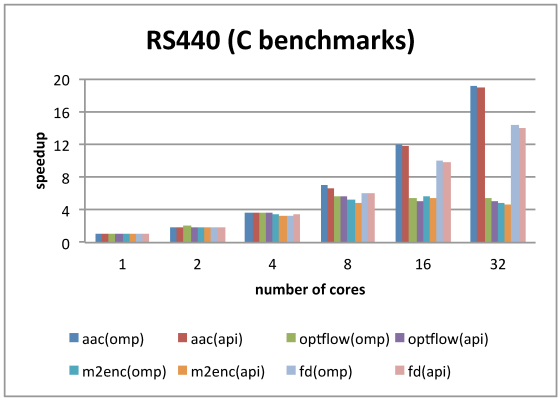

Fig. 12 Performance of C benchmarks on Hitachi Intel-core SMP RS440.

machines have similar performance results for the OpenMP and OSCAR API Analyzer executables. Therefore, the API Analyzer allows the set of C benchmarks to achieve with sequential compilers a similar performance compared to the one achieved with OpenMP compilers.

4.2 Embedded systems

Experimental setup

Table 4.2 shows the hardware details of the embedded multicores for the experiments. The table also shows the sequential compiler that is used with OSCAR API Analyzer. All the applications have been compiled with the maximum optimization level for the compiler, unless otherwise specified.

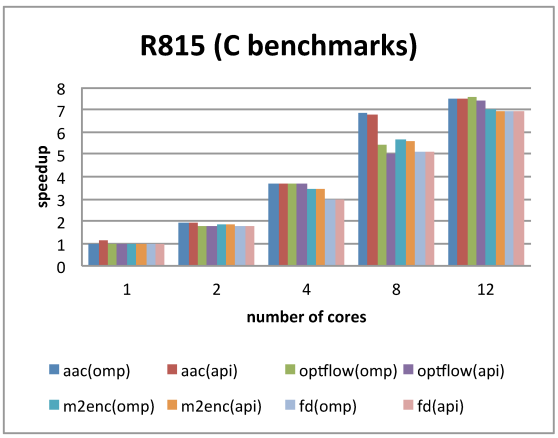

Fig. 13 Performance of C benchmarks on Dell PowerEdge AMD-core SMP R815.

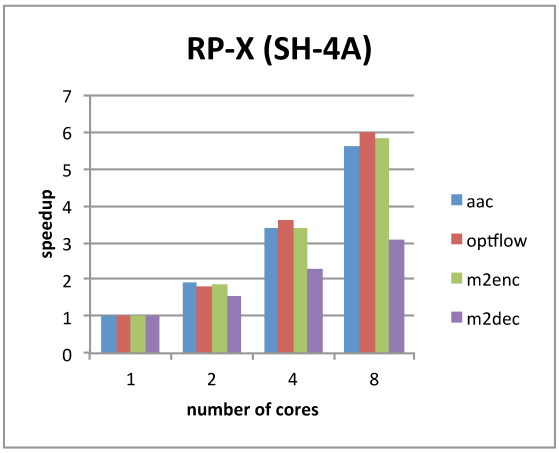

Fig. 14 Performance of benchmarks on RP-X multicore.

Homogeneous embedded systems

All the benchmarks used in the evaluation of the embedded homogeneous multicores are written in C. They are AAC encoder (`aac`), Optical Flow (`optflow`), MPEG2 encoder (`m2enc`) and MPEG2 decoder (`m2dec`) and Face detect (`fd`).

Figure 14 shows the performance on the heterogeneous multicore RP-X with no accelerators. `optflow` achieves the best speedup with 6x for 8 cores, similar to the speedups of `aac` and `m2enc`. However, `m2dec` has a poor scalability in this platform, because of the characteristics of slice level parallelism exploited by this application.

Tilera Tile64, in Figure 15, achieves good performance results with up to 41.5x for `aac` being executed on 64 cores.

From the 64-core TilePRO we move to the dual core Qualcomm Snapdragon S4. Its results in Figure 16 show that all the applications have more than 1.6x of speedup when executing on 2 cores, with up to 1.95x of speedup in the case of `aac`.

Finally, Figure 17 shows the results for the 3 ARM11-core ETKernel MPCore. The automatic parallelization of `aac` shows up to 2.85x of speedup. However, the performance of `optflow` with 3 cores is not improving compared to 2 cores, because of the nature of the computation of the application, which results in poor scalability for 3 cores.

Table 2 Evaluation environment (embedded systems)

System	RP-X	Tilera TilePro64	Qualcomm Snapdradon S4	ETKernel MPCore
CPU	SH-4A 648MHz (FE-GA 324MHz)	TLR36480(MIPS) 700MHz	Dual Krait 1.5 GHz	ARM 11 400 MHz
#cores	8 + 4 FE-GA	64	2	3
L1 Cache	16KB (ILRAM), 16KB (LDM), 64KB (DSM)	I$ 16KB, D$ 8KB	I$ 32KB, D$ 32KB	48KB
L2 cache	256 KB	64KB	1MB	-
Compiler	SH compiler	tile-gcc (gcc 4.4.3 based)	arm-linux-androideabi-gcc	RVDS 4.0

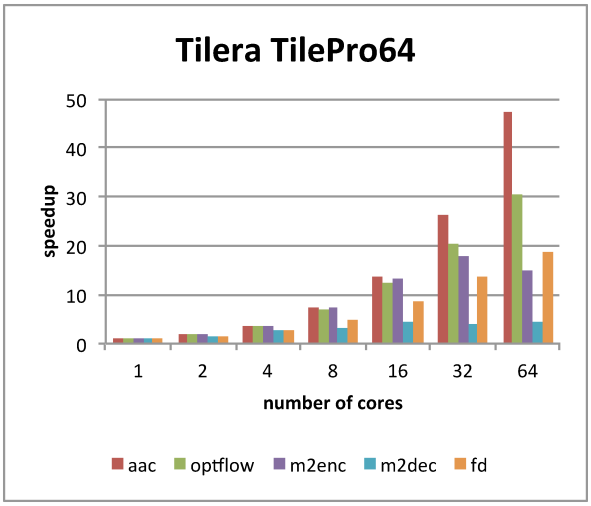

Fig. 15 Performance of benchmarks on Tilera Tile64.

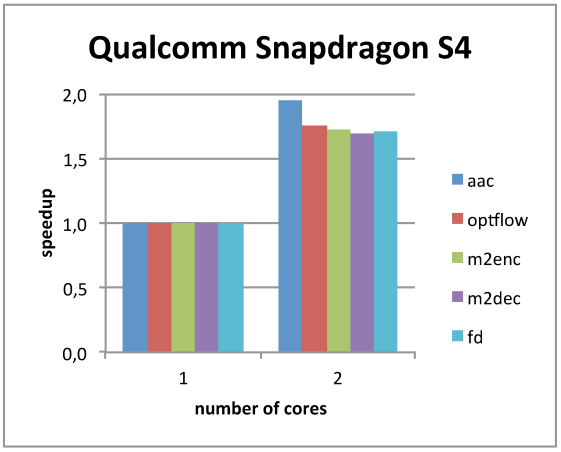

Fig. 16 Performance of benchmarks on Qualcomm Snapdragon S4.

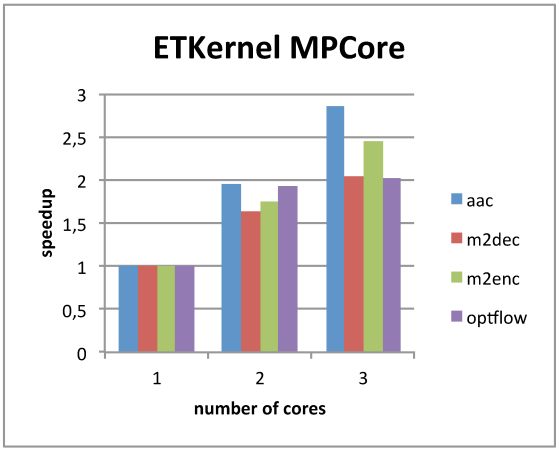

Fig. 17 Performance of benchmarks on ETKernel MPCore.

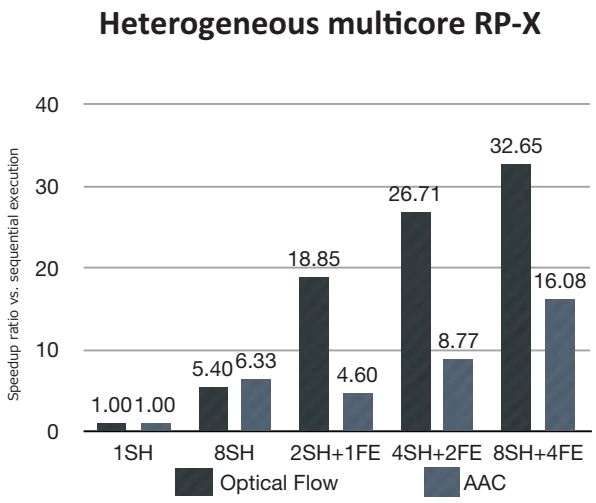

Fig. 18 Performance of benchmarks on the heterogeneous multicore RP-X with FE-GA accelerators

Heterogeneous embedded systems

The Heterogeneous Multicore RP-X is configured for these experiments with 4 FE-GA accelerators [14]. The code that is offloaded on the accelerators corresponds to a hand-tuned library code specifically for each application.

We evaluate its parallel performance with Optical Flow and AAC encoder for different configurations as we show in Figure 18.

The x-axis shows the processor configurations. For example, 8SH+4FE represents the configuration of eight SH-4A general-purpose cores and four FE-GA accelerator cores and the y-axis shows the speedup. The proposed framework achieved speedups of up to 32.65x with 8SH+4FE for Optical flow, and up to 16.08x for the AAC encoder. As we see, the optical flow benchmark benefits most of the accelerator cores, and we can observe super-linear speedups. The optical flow application uses vendor-supplied libraries for key-operations such as SADT (sum of absolute differences). Our own AAC-codec port is in its early stages and cannot yet fully exploit the potential of the accelerators.

5. Related work

Automatic parallelizing source-to-source compilers for homogeneous multiprocessors, such as Nanos Mercurium [2], SUIF [7] or Polaris [4] have proven to be a desirable path for the computing community interested in universalizing automatic parallelization. Multiplatform parallelizing compilers rely on multiplatform APIs for their deployment on several architectures. The Multicore Association works on proposals of multiplatform APIs for embedded systems, like the already released communication API [21]. OpenCL has been developed as a multiplatform parallel API [11],

mostly used on systems with accelerators like GPGPUs.

Par4All [22] is an automatic parallelizing and optimizing compiler for sequential programs written in Fortran and a strict subset of C. It generates OpenMP, CUDA and OpenCL code, and can be coupled with an accelerator run-time that provides the adaptation layer for different accelerators.

The compiler framework proposed by Lee [15] translates standard OpenMP shared-memory programs into CUDA-based GPGPU programs. Other work that targets heterogeneous architectures is OMPSs [6]. Their proposed OpenMP based annotations that ease programmability. Experiments with their compiler infrastructure generate parallel programs for shared-memory servers as well as Cell BE and GPUs. One more compiler for heterogeous multicores, CellSs [3], performs automatic parallelization of a subset of sequential C programs with data flow annotations on the Cell BE architecture.

The `cafc` [5] Fortran parallelizing compiler target several cluster platforms. It is based on the Co-array Fortran global address space programming model. Authors evaluate a set of benchmarks on several cluster architectures and compare their results to those of MPI, as we have compared ours to OpenMP.

6. Conclusions and Future work

Although automatic parallelizing compilers that address platforms as diverse as servers or embedded systems are scarce, their possibilities are promising in a world where multicores will dominate for several generations. To overcome the difficulty of developing parallelizing compilers for every platform, new strategies of compilation have to be proposed.

In this paper, we have introduced the OSCAR API Analyzer, a support set of tools and libraries that enable OSCAR compiler to be used for automatic parallelization just using sequential compilers. The API Analyzer enables easy portability, which is important for embedded systems, where OpenMP support is usually not available. The compilation framework presented generates parallelized code for shared-memory multiprocessos as well as homogeneous and heterogeneous embedded multicores. It also allows the rapid development of parallelizing compilers for new platforms like the heterogeneous multicore RP-X, minimizing the implementation costs of OpenMP compilers for each multicore aimed to be supported with automatic parallelization. We evaluated the framework with C and Fortran benchmarks on different platforms. Server results show up to 18x in 32 cores for the swim2000 benchmark on Hitachi SR16000 using the OSCAR API Analyzer. Furthermore, performance and scalability using sequential compilers as the backend compilers with OSCAR API Analyzer are similar to using the OpenMP compiler as the backend compiler for OSCAR API programs. Therefore, the Analyzer is capable of the same performance as the highly-tuned OpenMP support of the vendor compilers. For homogeneous embedded systems, we observed in the AAC encoder benchmark speedups up to 47x on the 64-core machine TilePro64. On the other hand, for the heterogeneous multicore RP-X, the Optical flow benchmark achieve up to 32.65x of speedup for 8 cores and 4 FE-GA accelerators. To our knowledge, this is the first compiler infrastructure of this kind to present a so detailed study in such a wide type of systems.

However, there are some aspects that have been left for future work, such as the power analysis control capabilities of the OSCAR API Analyzer, or the evaluation of its performance using other sequential compilers as backend, like XLC or ICC for several server machines.

Acknowledgments The authors would like to thank the OSCAR API Committee for their valuable support in this research.

References

[1] M. Amini, B. Creusillet, and S. Even. *Par4All: From Convex Array Regions to Heterogeneous Computing*. 2nd International Workshop on Polyhedral Compilation Techniques (IMPACT 2012), 2012.
[2] J. Balart and A. Duran. *Nanos mercurium: a research compiler for openmp*. European Workshop on OpenMP, 2004.
[3] Bellens, P., Perez, J. *CellSs: a programming model for the Cell BE architecture*. SC 2006 Conference, 2006
[4] Blume, W., Doallo, R., and Eigenmann, R. *Parallel programming with Polaris*. Computer, (1996).
[5] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. *A multi-platform Co-Array Fortran compiler*. Proceedings of the 13th Intl. Conference of Parallel Architectures and Compilation Techniques, 2004.
[6] R. Ferrer, J. Planas, and P. Bellens. *Optimizing the exploitation of multicore processors and GPUs with OpenMP and OpenCL*. Languages and Compilers for Parallel Computing, pages 215–229, 2011.
[7] M. Hall and J. Anderson. *Maximizing Multiprocessor Performance with the SUIF Compiler*. Computer, pages 84–89, 1996.
[8] A. Hayashi, Y. Wada, T. Watanabe, T. Sekiguchi, M. Mase, J. Shirako, K. Kimura, and H. Kasahara. *Parallelizing compiler framework and API for power reduction and software productivity of real-time heterogeneous multicores*. Languages and Compilers for Parallel Computing, pages 184–198, 2011.
[9] H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. *A multi-grain parallelizing compilation scheme for OSCAR (optimally scheduled advanced multiprocessor)*. Languages and Compilers for Parallel Computing, pages 283–297, 1991.
[10] H. Kasahara, M. Obata, and K. Ishizaka. *Automatic coarse grain task parallel processing on SMP using OpenMP*. Languages and Compilers for Parallel Computing, pages 189–207, 2001.
[11] Khronos Group. OpenCL. http://www.khronos.org/opencl.
[12] K. Kimura, Y. Wada, H. Nakano, T. Kodaka, K. Ishizaka, and H. Kasahara. *Multigrain Parallel Processing on Compiler Cooperative Chip Multiprocessor*. 9th Annual Workshop on Interaction between Compilers and Computer Architectures (INTERACT'05), pages 11–20, 2005.
[13] K. Kimura, M. Mase, H. Mikami, T. Miyamoto, J. Shirako and H. Kasahara. *OSCAR API for real-time low-power multicores and its performance on multicores and SMP servers*. Languages and Compilers for Parallel Computing, pages 188–202, 2010.
[14] T. Kodama, T. Tsunoda, M. Takada, H. Tanaka, Y. Akita, M. Sato, and M. Ito. *Flexible Engine: A Dynamic Reconfigurable Accelerator with High Performance and Low Power Consumption*. COOL Chips IX, 2006.
[15] S. Lee, S. Min, and R. Eigenmann. *OpenMP to GPGPU: a compiler framework for automatic translation and optimization*. ACM Sigplan Notices, pages 101–110, 2009.
[16] M. Mase, Y. Onozaki, K. Kimura, and H. Kasahara. *Parallelizable C and Its Performance on Low Power High Performance Multicore Processors*. Proc. of 15th Workshop on Compilers for Parallel Computing, 2010.
[17] *OpenMP. Simple, Portable, Scalable SMP Programming.* http://www.openmp.org/.
[18] *OSCAR API v2.0*. http://www.kasahara.elec.waseda.ac.jp/api2/regist.html.
[19] J. Shirako, N. Oshiyama, and Y. Wada. *Compiler control power saving scheme for multi core processors*. Languages and Compilers for Parallel Computing, 2006.
[20] Standard Performance Evaluation Corporation. *SPEC CPU2000*.
[21] The Multicore Association. *Multicore Communication API (MCAPI)*.
[22] M. Torquati, M. Vanneschi, and M. Amini. *An innovative compilation tool-chain for embedded multi-core architectures*. Embedded World Conference, 2012.
[23] Yuyama, Y., Ito, M., Kiyoshige, Y., Nitta, Y., Matsui, S., Nishii, O., Hasegawa, A., et al. *A 45nm 37.3GOPS/W Heterogeneous Multi-Core SoC*. ISSCC, pages 86-87, 2010.