Automatic Design Exploration Framework for Multicores with Reconfigurable Accelerators

Cecilia González-Álvarez[1,2], Haruku Ishikawa[1], Akihiro Hayashi[1], Daniel Jiménez-González[2], Carlos Álvarez[2], Keiji Kimura[1], and Hironori Kasahara[1]

[1] Waseda University
[2] Universitat Politècnica de Catalunya
{cecilia,iharuku,ahayashi}@kasahara.cs.waseda.ac.jp,
{djimenez,calvarez}@ac.upc.edu,
kimura@apal.cs.waseda.ac.jp, kasahara@waseda.jp

Abstract. Despite their promising improvements in performance and power efficiency, the possibilities of reconfigurable computing in multiprocessor environments are still mainly unexplored. This paper presents a new framework for rapid testing of multicores with application-specific reconfigurable accelerators. The design exploration is a multi-step flow that performs automatic generation of fine-grained accelerators, automatic parallelization, and testing. The automatic generation of accelerators is accomplished by an in-house developed software. Automatic parallelization is performed by OSCAR (Optimally SCheduled Advanced multiprocessoR) source-to-source compiler. The testing platform is configured from 1 up to 8 cores, and each core has a private reconfigurable space to implement application-specific accelerators. We test the design process with applications from the media domain (Optical Flow, AAC encoder, MPEG2 encoder and MPEG2 decoder). The results show that the multicore configuration outputs of our design exploration framework for 8 CPUs with reconfigurable accelerators achieve a maximum speedup of 6.57x for the Optical Flow application.

1 Introduction

Recently, heterogeneous multicores have emerged as a solution for the high-performance and low-power market [1]. However, the enormous range of possibilities of heterogeneous multicore systems makes it difficult to join the easiness to program of homogeneous systems and the performance and power savings of heterogeneous multicores. Besides, reconfigurable technologies for application-specific accelerators have shown performance improvement and flexibility of design at low cost. The addition of reconfigurable hardware to such systems increases the challenge of designing and programming them. Therefore, design exploration of new reconfigurable heterogeneous architectures is necessary. However, it requires considerable programming effort [2].

To address the aforementioned problems, we propose an automatic design exploration framework for a reconfigurable heterogeneous multicore architecture.

Our proposal helps to: significantly reduce the programming effort by automatically generating reconfigurable accelerators, and automatically integrate those accelerators into the target application. This integration is done in such a way that OSCAR compiler [3] can auto-parallelize and synchronize them with the rest of tasks of the application. The target architecture is a reconfigurable heterogeneous multicore architecture that uses tightly coupled reconfigurable units in an otherwise homogeneous multicore to join the best of the two worlds. The reconfigurable heterogeneous system is easily and automatically programmable with the help of the OSCAR compiler. At the same time, the use of a new automatic framework to program the reconfigurable units takes profit of the performance benefits of such units. Although the use of reconfigurable functional units has been previously studied for uniprocessors [4–7], this work explores the synergies between reconfigurable accelerators and multicores. OSCAR API [8] supports OSCAR parallelizing compiler and the target multicore, and it also supports the proposed framework as a bridge between the accelerator compiler and OSCAR compiler; thus we can easily explore the design space for accelerated multicores.

Therefore, the main contributions of this work are: (1) rapid validation framework for testing of design alternatives for reconfigurable accelerators; (2) testing of the automatic parallelization in reconfigurable multicore architectures with accelerators automatically generated; (3) exploration of the feasibility of multiprocessor cores with reconfigurable tightly-coupled accelerators for applications of the media domain.

The rest of the paper is organized as follows. First, Section 2 describes the heterogeneous multicore architecture that this paper targets. Section 3 defines the framework for the exploration of new architecture designs and explains the methods and tools involved. Section 4 establishes the experimental setup for the results presented in Section 5. We finalize in Section 6 with the conclusions.

2 Target Architecture

The target multicore architecture is based on the OSCAR architecture abstraction [9]. It is composed of general-purpose processor cores with tightly-coupled reconfigurable accelerators. Each one of the cores in the multicore architecture can be seen as an OpenSPARC-based core with a run-time reconfigurable functional unit that can implement several configurations of application-specific hardware accelerators [7]. The reconfigurable accelerator is integrated in the general-purpose processor pipeline as an additional functional unit; therefore it accesses the general register file. Data memory transfers are handled by the general-purpose processor memory controller. Additionally, the reconfigurable accelerator has an internal register file with a flexible number of configurable entries to allow computations with a variable degree of instruction parallelism. We use special instructions added to the general-purpose processor instruction set to control the accelerator execution. We also define three extra instructions to move multiple data from the processor register file to the accelerator, and another instruction to move accelerator output data of arbitrary size back to the processor registers, which enables multiple input and multiple output computations.

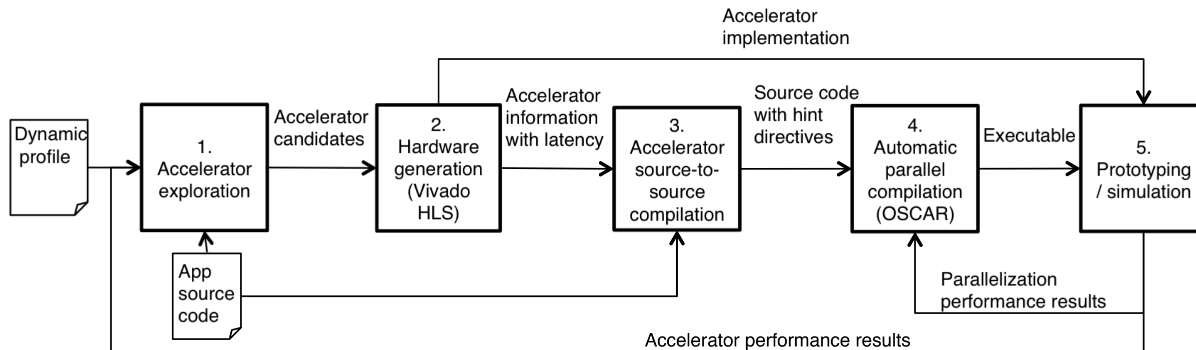

Fig. 1. Framework for the design of reconfigurable accelerators in a heterogeneous multiprocessor.

3 Design exploration and automatic parallelization

The design exploration targets the architecture of multicores with a reconfigurable area for hardware accelerators described in Section 2. The exploration enables rapid prototyping of the accelerators relying on tools for automation. We use a feedback-based approach for constant improvement of the design. Reconfiguration gives the flexibility to adapt the design to different applications. In our proposal programmers do not rewrite the applications because no specific programming model is needed.

Figure 1 shows the high-level diagram of our framework implementation. The *Accelerator explorer* (box 1 in Figure 1) is implemented on top of LLVM compiler [10]. Its inputs are the source code and the dynamic execution profile of the application. As a result of this step, we get those parts of the application that can be executed as custom instructions on a new accelerator. The *Hardware generator* (box 2) uses information about the previous accelerator candidates to generate the accelerator implementation in a hardware description language (HDL), and to obtain the latency information. For this step, we rely on the proprietary Vivado High-Level Synthesis (HLS) tool from Xilinx driven by our code. The *Accelerator source-to-source compiler* (box 3) performs a source-to-source compilation of the application based on the accelerator description and the accelerator latency. After this step, we can pass on the transformed application code for parallelization to the *Automatic parallelizing compiler* (box 4). Here, the source code includes hint directives, which provide information about the accelerators, such as the accelerable code fragments and their execution cost as shown in Section 3.4, which are required for the scheduling phase in the compiler. We use the OSCAR parallelizing compiler to schedule the parallel tasks on the CPUs and the custom instructions on the accelerators. The OSCAR compiler automatically parallelizes C or Fortran77 programs on various multicore processors with the help of OSCAR API[8, 3], which is an extended subset of OpenMP. The

compiler outputs an executable to run on the *Prototype/simulator* (box 5), that is configured with the accelerator implementation information. The execution of the application with accelerators may provide feedback about the performance results to the *Accelerator explorer* and the *Automatic parallelizing compiler*.

In the coming subsections, we explain details about some of the design exploration steps.

3.1 Accelerator exploration

The accelerator explorer objective is to analyze the target application source code to provide a list of the most promising accelerators to be used in the target architecture of choice. It is divided into the *Analysis* and *Selection* phases.

Analysis This phase analyzes the source code of the application to enumerate the groups of instructions that can be scheduled together as a new instruction to be executed on a hardware accelerator.

The objective is to find all the maximal valid subgraphs[3] of a data flow graph for a given basic block. These subgraphs include only instructions that can be executed in the accelerators and that satisfy the convexity constraint. We exclude memory and branch instructions since they are regarded as *invalid* instructions in the search. The search is done with a fast implementation of the algorithm presented by Li *et al.* [11] using binary structures.

Selection The selection phase provides a list of possible accelerators, ordering them by their likeliness to improve the speedup of the application. We developed a heuristic selection of the best accelerators among the subgraphs of groups of instructions. The heuristics are based on the profile data of the dynamic execution of the application, the frequency of execution of the basic block and the latency gain estimation, which is derived from the critical path in cycles obtained in the Vivado HLS synthesis.

3.2 Accelerator generation

The accelerator generation converts the direct acyclic graph representations of the most promising accelerators into their RTL implementation.

In a first step, we generate the equivalent C code of each accelerator with the extra information needed for the C to HDL conversion. The resulting code is passed to the Vivado HLS tool, which transforms that code into the RTL design that implements the accelerator in hardware. From the real hardware implementation, we obtain the latency of the design that we are going to use in the integration with the OSCAR compilation framework for heterogeneous architectures. That latency does not include the communication of the accelerator inputs and outputs. We calculate the total latency (communication of data and computation), with the following formula:

$$Total_{lat} = Base_{lat} + T_{mem} + (N/2) + M$$

[3] A subgraph S of a graph G is maximal and valid if it is the largest possible subgraph that does not violate some given restrictions, for instance, the convexity constraint.

```
void optflow_main_loop_bb10_1 ( int *o0, int i2, int i3, int i1, int i0) {
    *o0= i2<i0-1&&i3<i1-1;
}
```

(a) Accelerator function stub definition

```
#pragma oscar_hint accelerator_task (ACC_0) cycle(5)
optflow_main_loop_bb10_1( &(main_flag), i, j, NUMBERBLOCKS_X, NUMBERBLOCKS_Y);
```

(b) OSCAR hint directive and call to function stub

Fig. 2. Application codes with OSCAR hint directives. (a) Encapsulated code of the part of the application that is marked to be accelerated. (b) OSCAR hint directive with the accelerator latency, 5 for this accelerator, and the call to the stub that defines the accelerator behaviour.

where $Base_{lat}$ is the latency of the design from Vivado HLS, T_{mem} is the measurement of the average memory transfers latency, and N and M are the number of inputs and outputs of the accelerator, respectively. As it can be derived from the formula, we suppose a 2 read ports and 1 write port bank register.

3.3 OSCAR integration

The original code of the application is source-to-source compiled to include the generated accelerators in a way that can be recognized by the OSCAR compiler.

To accomplish that, for every accelerator, first we encapsulate the involved code in a function stub. Second, we substitute the code that targets the accelerator by an OSCAR hint directive with the calculated accelerator latency, followed by a call to the function stub with the appropriate parameters. Figure 2 shows an example of this substitution.

3.4 OSCAR parallelizing compiler

The OSCAR parallelizing compiler takes a source code with hint directives and automatically generates a parallelized executable for the prototyping platform as shown in Figure 1.

Hint Directives for OSCAR Compiler OSCAR hint directives before a function [3] indicate that the specified function can be executed on the specified accelerator in the given latency. For example, the directive *#pragma oscar_hint accelerator_task (ACC_0) cycle(5)* in Figure 2.b tells the OSCAR compiler that the function *optflow_main_loop_bb10_1* takes 5 cycles on the accelerator called ACC_0. This latency information is used in the task scheduling scheme of the OSCAR compiler. Description of that task scheduling follows.

Multigrain Parallel Processing and Task Scheduling The OSCAR compiler decomposes a program into coarse grain tasks, namely macro-tasks (MTs), such as basic block (BPA), loop (RB), and function call or subroutine call (SB) hierarchically. OSCAR compiler can exploit not only loop level parallelism, but also coarse-grain task parallelism and near fine-grain statement-level parallelism [9].

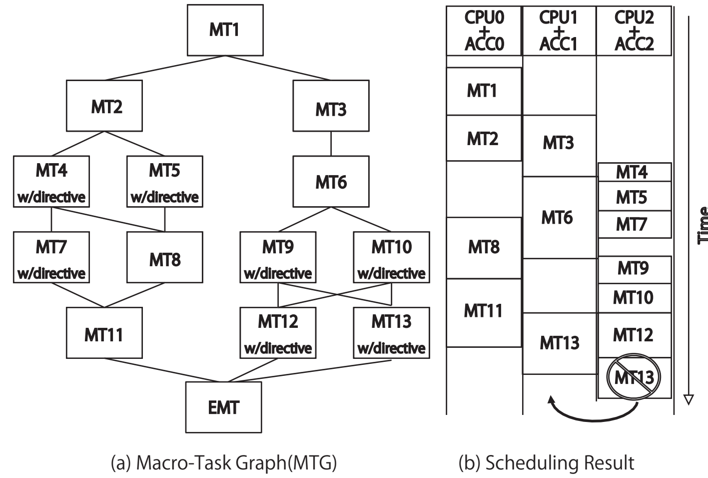

Fig. 3. Macro-Task Scheduling Scheme for Heterogeneous Multicore. Macro-tasks in Subfigure (a) are mapped by OSCAR compiler in Subfigure (b) to the processors considering data dependencies; the compiler estimates the latency of each task, except for the accelerator latency given by hint directives.

After that, the OSCAR compiler analyzes control flow and data dependencies among macro-tasks and generates macro-flow graphs (MFG). MFGs are transformed to macro-task graphs (MTG), which show coarse-grain parallelism, after earliest executable condition analysis [12, 13].

Then, the OSCAR compiler maps all macro-tasks to the processors statically, considering the overhead for data-transfers and synchronizations [14]. Figure 3(a) shows an example of a macro-task graph with some macro-tasks that contain accelerator hint directives. Figure 3(b) shows the scheduling result in a system with three CPUs (CPU0-CPU2), each one of them with their own accelerator (ACC0-ACC2). First, the compiler assigns initially ready MT1 to CPU0+ACC0. After that MT2 and MT3 are assigned to CPU0+ACC0 and CPU1+ACC1 respectively, since MT2 and MT3 are ready after the execution of MT1 on CPU0+ACC0 finishes. Similarly, the compiler assigns each task until all tasks have been assigned. The OSCAR compiler is aware of the performance gain with accelerators by referring the hint directives information, and otherwise estimates the latency in the CPUs from the middle-path intermediate representation information.

4 Experimental Setup

For the results presented in this paper, our framework uses a software-based cycle-accurate simulator called *tomato* for rapid testing. *tomato* simulates the execution of an input application on a shared-memory multicore architecture with the possible addition of accelerators. It can be configured with different ar-

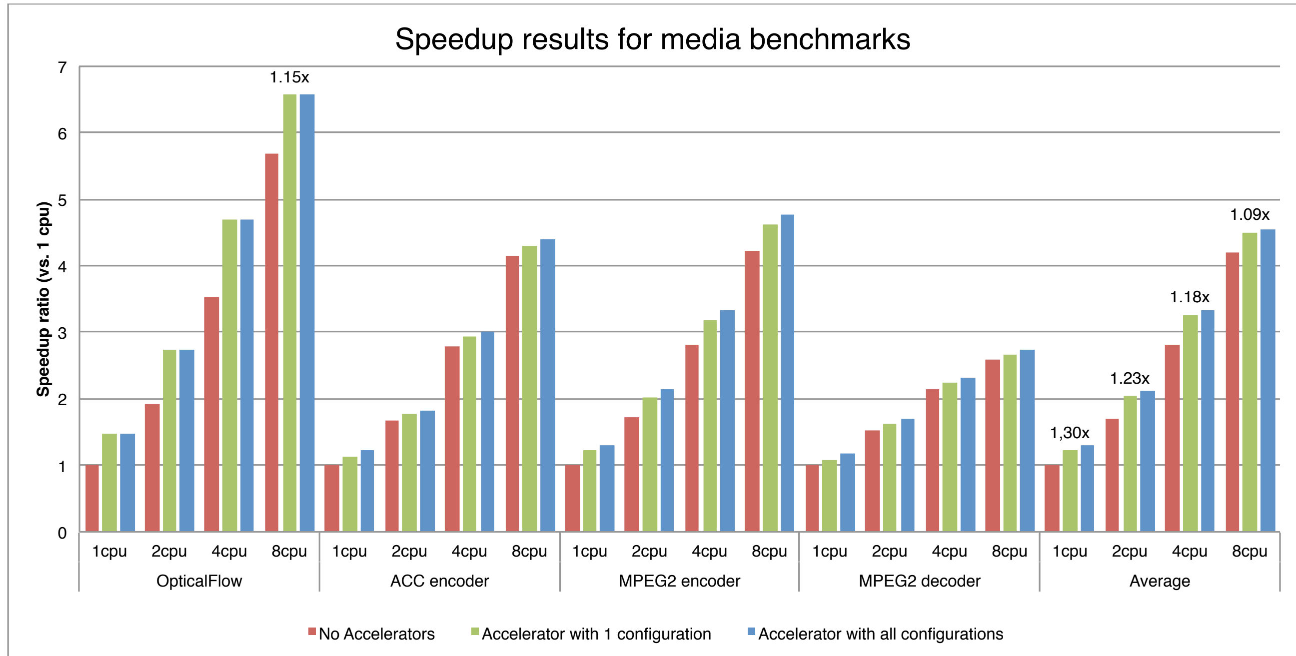

Fig. 4. Speedup results for the media benchmarks.

chitecture parameters such as number of processors, instruction set architecture, memory model, memory latency and interconnection network type.

In this paper, we use *tomato* with the SPARC V9 instruction set. The number of cores in the experiments ranges from 1 to 8. The centralized shared memory size is 1 GB, and its latency is 60 cycles. Each general purpose processor core is configured with a snoop cache. L1 cache size is 32 KB, and its latency is 1 cycle, while L2 cache size is 512 KB, with 4 cycles latency. Codes where cross-compiled for SPARC with gcc 4.2.4, optimization level flag -O3.

5 Results

In this section, we evaluate the proposed reconfigurable multicore architecture framework. In particular, we show the performance results obtained for a set of applications with automatically obtained configurations. Therefore, we provide in this evaluation not only results for just sequential accelerated applications, but specially for various parallelized applications that use accelerators.

The C benchmarks used in this evaluation are AAC encoder, Optical Flow, MPEG2 encoder and MPEG2 decoder. These benchmarks are from known benchmark suites such as Mediabench [15] and OpenCV [16].

Figure 4 shows the performance results for the benchmarks in several multicore configurations. The horizontal axis is divided by application. At last it

shows the average values of all the performance results. For each application, the horizontal axis shows the number of CPUs targeted in the parallelization, from 1 to 8. For each CPU group, we show 3 type of results: CPUs without accelerators (red bar), CPUs with accelerators with only 1 configuration (the most promising one) fixed at the beginning of the execution (green bar), and CPUs with accelerators that use all the configurations found (blue bar). This later case may be only possible in a system with reconfiguration at execution time. The vertical axis shows the speedup ratio. The speedup baseline is 1 CPU without accelerators. Note that when accelerators are present all CPUs have their own accelerator. For instance, in a system with 8 CPUs, we have 8 accelerators, one attached to each CPU. Numbers above the bars indicate the relative speedup between the system without accelerators and the system with all the accelerator configurations. The total number of configurations depends on the application: 3 different configurations in Optical Flow, 20 in AAC encoder, 3 in MPEG2 encoder and 8 in MPEG2 decoder.

The best results are for the Optical Flow application with a maximum speedup of 6.57x for 8 CPUs with all the accelerator configurations. For this application, the speedup for 8 CPUs with accelerators is 6.57x, 1.15x more than the 8 CPU speedup with no accelerators (5.69x). For this concrete application, the speedup difference between each configuration with and without accelerators is 1.35x in average. In the case of AAC encoder, though, the best speedup ratio between the non-accelerated and the accelerated version is 1.22x. This is because the floating-point burden of the computation is not suitable for the reconfigurable silicon we target. MPEG2 encoder has a speedup of 4.78x for 8 CPU + 8 accelerators, 1.12x more than the 4.23x result of the 8 CPU version without accelerators. MPEG2 decoder application shows poor scalability in the regular parallelization because of the characteristics of slice level parallelism exploited by this application. The accelerator version is influenced also from this parallelization issue, and its speedup compared to the non-accelerated version goes from 1.18x (1 CPU) to 1.09x (8 CPU). Finally, the average numbers show a speedup up to 4.56x for 8 CPU with accelerators, in contrast with the 4.19x without accelerators for the same CPU configuration. The average relative speedup for all applications and all the multicore variations is 1.2x. Note that all the speedups achieved with accelerator configurations require no effort to the programmer because the accelerator configurations and the hint directives for OSCAR compiler are generated automatically.

The results presented above depend highly on the total percentage of execution time of the application where the accelerators are being used. For the MPEG2 encoder, the average speedup of the code fragments that executes on accelerators is 12.51x, while for the Optical Flow benchmark it is a mere 5x. However, Optical Flow speedup results are better because its accelerators target 40% of the code, whereas the MPEG2 encoder ones cover only 25.71% of the code. Besides, the AAC encoder has a 6.35x speedup in the 21.28% part of the code that is accelerated. In the case of the MPEG2 decoder, only 3.04x of speedup is obtained in 21.28% of accelerated part. These results lead to think

about Amdahl's law, since a more intensive use of accelerators will result in an overall speedup increase. Therefore, future efforts should be directed towards improving the accelerators capabilities to cover more parts of the application that for this work were excluded, as well as taking into account data-level parallelism for super-linear speedups.

The results regarding the number of configurations used differ depending on the application. Optical Flow has a great performance improvement with just one accelerator added (green bar against blue bar). However, applications AAC encoder and MPEG2 encoder can benefit from having more than only 1 accelerator. In that case, a fast reconfigurable accelerator will be desirable.

6 Conclusions

This paper has proposed an automatic design exploration framework for multicore architectures with tightly-coupled reconfigurable accelerators automatically generated. We can use the framework for rapid prototyping and constant improvement of the design through feedback mechanisms. The automation both in the generation of accelerators and in the parallelization with OSCAR compiler allows the rapid prototyping to be effective. Furthermore, to our knowledge, this is the first work that explores the use of a reconfigurable unit in a multicore environment.

We have evaluated the framework with benchmarks from the media domain. Results show a speedup up to 6.57x for the Optical Flow benchmark with 8 CPUs and accelerators, while the same configuration of CPUs without accelerators gives 5.69x of speedup. In average, for all the evaluated applications, we reach a speedup up to 4.56x for 8 CPU with accelerators, in contrast with the 4.19x without accelerators. It is noteworthy that the addition of accelerators gives performance improvements over the version without accelerators that scale regardless of the number of cores considered. This fact highlights the importance of adding reconfigurable units to heterogeneous multicores as they provide new, orthogonal gain to the system, otherwise impeded by Amdahl's law.

As a future work we plan to improve the performance obtained with the accelerators, although the speedup gain will depend on the implicit parallelism in the code, as well as on the workload that impacts the data transfer cost. Other future research direction includes the analysis of power reduction in the presented architecture. Finally, we consider that in a future work we should also include an accelerator scheduling aware of reconfiguration time, instead of the idealized solution presented here.

Acknowledgments

This work is supported by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contracts TIN2007-60625 and CSD2007-00050, by the Generalitat de Catalunya (contract 2009-SGR-980), and by the Japanese Ministry of Technology and Education (Monbukagakusho scholarship). We would also like to thank the Xilinx University Program for its hardware and software donations.

References

1. The Green500 List November 2012. Heterogeneous systems re-claim green500 list dominance. http://www.green500.org/lists/green201211, 2012.
2. Jason Cong, Vivek Sarkar, Glenn Reinman, and Alex Bui. Customizable Domain-Specific Computing. *IEEE Design & Test of Computers*, 28(2):6–15, March 2011.
3. Akihiro Hayashi, Yasutaka Wada, Takeshi Watanabe, Takeshi Sekiguchi, Masayoshi Mase, Jun Shirako, Keiji Kimura, and Hironori Kasahara. Parallelizing compiler framework and api for power reduction and software productivity of real-time heterogeneous multicores. In *The 23rd International Workshop on Languages and Compilers for Parallel Computing (LCPC2010)*, Oct. 2010.
4. Scott Hauck and TW Fry. The Chimaera reconfigurable functional unit. *IEEE Transactions on VLSI Systems*, 12(2):206–217, 2004.
5. Jesse Benson, Ryan Cofell, Chris Frericks, and CH Ho. Design Integration and Implementation of the DySER Hardware Accelerator into OpenSPARC. *Proceedings of 18th*, (Section 3), 2012.
6. Jonathon Evans, Kyle Rupnow, and Katherine Compton. Reconfigurable Functional Units for Scientific Superscalar Processors. *2007 International Conference on Field-Programmable Technology*, pages 73–80, December 2007.
7. Cecilia González-Álvarez, Mikel Fernández, Daniel Jiménez-González, Carlos Alvarez, and Xavier Martorell. Automatic Generation and Testing of Application Specific Hardware Accelerators on a New Reconfigurable OpenSPARC Platform. In *Workshop in Reconfigurable Computing, HiPEAC*, number 1, 2011.
8. K. Kimura, M. Mase, H. Mikami, T. Miyamoto, and J. Shirako H. Kasahara. Oscar api for real-time low-power multicores nad its performance on multicores and smp servers. *Proc of The 22nd International Workship on Languages and Compilers for Parallel Computing(LCPC2009)*, 2009.
9. Keiji Kimura, Yasutaka Wada, Hirofumi Nakano, Takeshi Kodaka, Jun Shirako, Kazuhisa Ishizaka, and Hironori Kasahara. Multigrain parallel processing on compiler cooperative chip multiprocessor. In *Proc. of 9th Workshop on Interaction between Compilers and Computer Architectures (INTERACT-9)*, Feb. 2005.
10. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, (c):75–86.
11. T. Li, Z. Sun, W. Jigang, and Xicheng Lu. Fast enumeration of maximal valid subgraphs for custom-instruction identification. In *Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 29–36. ACM, 2009.
12. H. Honda, M. Iwata, and H. Kasahara. Coarse grain parallelism detection scheme of a fortran program. *Trans. of IEICE*, J73-D-1(12):951–960, Dec. 1990.
13. H. Kasahara, H. Honda, A. Mogi, A. Ogura, K. Fujiwara, and S. Narita. A multigrain parallelizing compilation scheme for OSCAR (Optimally scheduled advanced multiprocessor). In *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, pages 283–297, August 1991.
14. Y. Wada, A. Hayashi, T. Masuura, J. Shirako, H. Nakano, H. Shikano, K. Kimura, and H. Kasahara. Parallelizing compiler cooperative heterogeneous multicore. In *Proceedings of Workshop on Software and Hardware Challenges of Manycore Platforms, SHCMP'08*, Jun. 2008.
15. Chunho Lee, Miodrag Potkonjak, and William H. Mangione-smith. Mediabench: A tool for evaluating and synthesizing multimedia and communications systems.
16. G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.