Turkish Journal of Electrical Engineering & Computer Sciences

http://journals.tubitak.gov.tr/elektrik/

Research Article

Turk J Elec Eng & Comp Sci
(2018) 26: 2595 – 2604

doi:10.3906/elk-1712-374

Incremental Banerjee test conditions committing for robust parallelization framework

Aimad Eddine DEBBI[1,2,*], **Haddi BAKHTI**[1]

[1]Department of Electronics, Faculty of Technology, Ferhat Abbas University, Setif, Algeria
[2]Department of Computer Science, Faculty of Mathematics and Informatics, Mohamed Boudhiaf University, M'sila, Algeria

Received: 28.12.2017 • **Accepted/Published Online:** 30.05.2018 • **Final Version:** 28.09.2018

Abstract: This paper describes the design of an automatic parallelization framework. The kernel supplied at its front end was suggested as an instrument for parallel potential assessment. It was used to measure the maximum achievable speedups in the major set of the CHStone benchmark suite programs. In such framework, we suggested the liberation of parallelism incrementally. We proposed a data dependency heuristic-based transformation method to make true dependences dissociation. We generated an internal representation (IR^2), where the Banerjee test conditions are met. Two among three of Banerjee test conditions came to be committed. In shared memory many/multicore platforms, the third condition could be satisfied by privatization. We would be able to choose the safe and the opportune pairwise (mapping-privatization) scheme among a number of threads mapping scenarios that become available in the IR^2 structure. Instrumentation on a subset of CHStone benchmark was carried out as a validity proof of our proposal, and the results confirmed that our framework kernel is robust.

Key words: Automatic parallelization, parallel programming, source-to-source compilation, data dependency profiling, parallelism assessment, benchmarking

1. Introduction

Demands for parallelizing frameworks are expected to increase considerably in the near future for two reasons. First, the popularity of multicore architectures is continuously growing, and second, the circuits have approached the physical barriers that are imposed by the frequency wall. A wide range of applications [1–3] have already been suggested within parallel implementations. Some parallelizations are carried out also using the semiautomatic platforms and the annotations/directives oriented frameworks like CUDA and OpenMP [4–7]. Although semiautomatic tools are often used for parallel implementation, they are not as easily applicable as it appears [8].

Full automatic parallelization tools are supposed to be reliable instruments for parallel implementations. However, their popularity is not at the desired level. It seems that they are not attractive enough. Thus, they need to be notably improved. Generally, parallelization frameworks [9–12] are source-to-source compilers. They involve mainly a number of modules to cover the generation of an Internal Representation (IR) after parsing, modules for profiling, analysis engines, modules for transformation passes, and finally rolling-back parsing modules to generate output sources. The most challenging tasks for parallelization tools are firstly, identifying

*Correspondence: aimad_ne@yahoo.fr

the parallel sections, and secondly, mapping these sections to Parallel Execution Units (PEU). We use PEU as a generic term. Nowadays, in shared memory many/multicore systems, PEU could simply be simultaneous threads that run on separate cores. They could also be some personalized parallel hardware if parallelization is targeting a high-level synthesis (HLS) for FPGA or if it targets specialized plate-forms.

Separation of parallel portions of programs is not an evident task since algorithms generally contain a lot of complex dependencies. Data dependences need to be profiled adequately to make parallelization successful. Several works [13–17] have already investigated data dependence profiling. In these works, data dependence profiling has not been addressed exclusively for automatic parallelization purposes. It has been investigated in a wide context of compilation optimizations. It has been addressed to deal with a number of optimizations such as partial redundancy elimination (PRE), runtime code scheduling [14], and performance tuning [16]. Integration of such profilers in autoparallelization tools seems to be challenging since some of them, particularly the dynamic data-dependence profilers [15–17], suffer from runtime overhead and memory overhead. Li et al. [16] stated that runtime overhead may elongate the analysis for several hours. Usually, those profilers use binary instrumentations and operate at the lower level of abstraction. In our opinion, this makes them not appropriate for integration in autoparallelization frameworks.

Mapping is the second critical task in automatic parallelization. Most works [18–21] adopted the common pairwise (threads, iterations-loops) mapping scheme for multicore architectures. Usually, they concentrate the dealing on loops regions and make loops iterations spread over threads. Some proposed techniques for making privatization successful [20–22], while others suggested threads speculations algorithms [18, 19, 23].

Unlike polyhedral tools [24], the parallelization compiler that we have built is not restricted to only certain code regions satisfying some exigencies and conditions. It does not focus only on some portions and constructs. The instrument we are suggesting allows the determination of parallel potential in all kinds of C programs. There are no restrictions or conditions about loops or their nests or subscripts. There is no focus only on particular regions. There are no prior suggestions or hypotheses about the code. The compiler we suggest here allows discrimination of parallelism inside deeply nested structures even if the inherent parallel potential is slight.

In privatization techniques, each thread maintains a copy of the privatized data. By such concept, we may avoid the conflict access to the shared space. However, in the mainstream parallelization techniques when we suggest the pairwise mapping, Thread–iteration-loop, the problem of privatizing data arises. Data cannot be privatized if its definition (*Def*) may reach its *Use* in other iterations, i.e. *Defs* in some threads may reach the corresponding *Uses* in other threads. Since we have commonly substantial loop-carried dependences in programs, data that cannot be privatized are very frequent in the pairwise thread–iteration-loop mapping scheme. Application of privatization in this case is too hard.

So, in this work, we suggested the liberation of parallelism incrementally in two steps. In the first step, we proposed a particular transformation that mitigates the major difficulties encountered in other tools. We obtained an appropriate structure notated IR^2 where we do not have to worry about all kinds of dependencies; instead, we have to deal with only the false dependences. False dependences alone do not raise serious problems if the adequate privatization scheme is chosen. In the second step, we selected the opportune mapping scheme among a number of mapping scenarios that become available thanks to this novel structure and we applied the convenient privatization. In this work, we will do the following:

- We give a brief description of the front-end part of our parallelizer. We clarify how parallelization would be made less hard using the concept of Incremental Conditions Committing (ICC).

- We illustrate how the kernel could be used as a reliable tool for theoretical parallel potential assessment. We instrument a subset of CHstone benchmark suite as a validity proof of our proposals and we expose the results as approximate estimations of the theoretical speedups.

2. Framework description

2.1. Framework architecture

State-of-the-art parallelization compilers like those provided in [10, 12, 25, 26] operate transformations to grant safety parallelization for a maximum possible number of loops. Results of transformations are either a set of annotations or some semantic modifications using optimization techniques such as induction variable reduction, reduction variable, constant folding, and dead code elimination. This makes loops less vulnerable to the conflicts in the pairwise thread–iteration-loop mapping scheme.

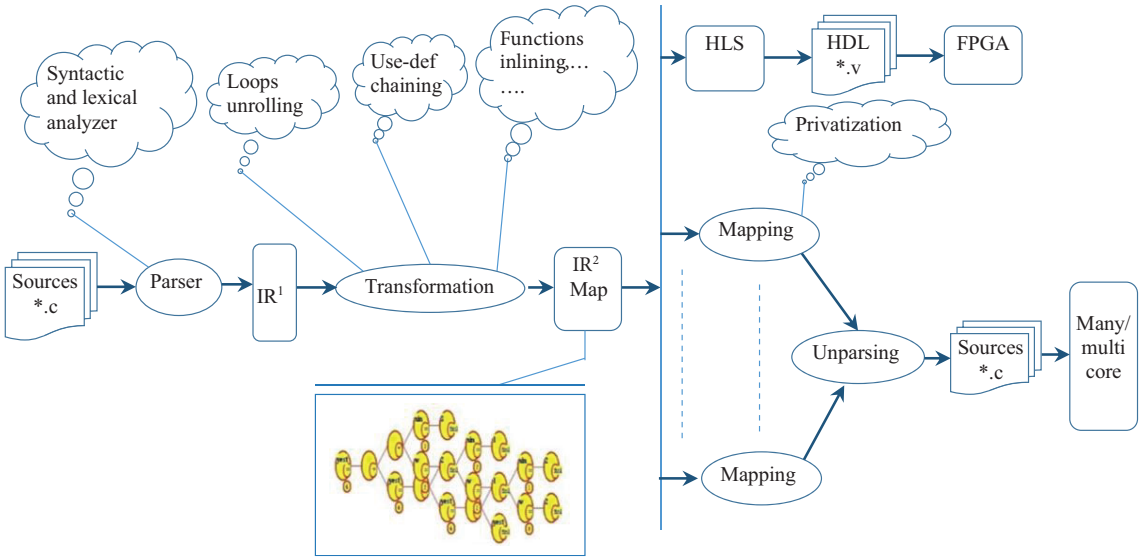

Figure 1. Framework architecture.

In the present investigation, we fundamentally used the simple concept of incremental conditions committing (ICC). We applied two transformations consecutively to get what we call "the *Map*", where two among three of Banerjee test conditions [27] are met. These two committed conditions are: first, the No flow dependence. Flow dependence is also referred to as Read After Write (RAW) dependence. Second, the No output dependence. Output dependence is also referred to as Write After Write (WAW) dependence. These two dependences are considered to belong to what is called the true dependences class. The third condition of Banerjee test is No Write After Read (WAR) or no antidependence. WAR is considered to belong to the false dependences class. This third condition would be satisfied by combining the privatization technique with an appropriate mapping. Privatization here would not be subject to the problems encountered in other frameworks since we performed a particular mapping and not the classical pairwise mapping thread–iteration-loop.

Figure 1 summarizes the design logic of our compiler. After being parsed, the program is translated to an Abstract Syntax Representation (ASR) also referred to as first internal representation IR^1. It then undergoes a second particular transformation where a total data dependency resolution it is carried out. Only the false dependences remain in the resultant structure. The outcome is a set of directed graphs which are considered

the second internal representation IR^2 and are called "the *Map*".

2.2. Work-flow

The construction of the *Map* is guided mainly by the data dependency heuristic. We fundamentally used the *use-def* chaining. A number of techniques, especially function in-lining, loops unrolling, aliasing, constant and copy propagation among others, are imperatively needed. Other analysis and optimization techniques commonly involved in other compilers were discarded because their effect would be neutralized in the suggested transformation process. *Use-def* chaining is applied from the top to the bottom for each *def* in the whole sets of program *defs* to carry out a reduced equivalent set. In this reduced set, there was not any true dependence between any pair of *defs*. So, the two Banerjee test conditions [27] RAW and WAW were met. The *Map* is a set of directed graphs that express the program outputs and the whole processing involved for their production. It is also a representation of the equivalent set of *defs* previously produced by the *use-def* chaining. Figure 2 shows a partial result of the *Map* that was generated by the compiler on instrumenting Algorithm 1 as an input program.

Algorithm Adaptive filtering kernel
(as an arbitrary algorithm example involving nested loops-carried dependencies)

a) The generic pseudo algorithm

1: $for\ k = k_0 : k_0 + N$
2: $\quad Y_{estimated} = W^{top} * X_{input}$
3: $\quad err = Y_{estimated} - Y_{wanted}$
4: $\quad W = W + \mu m\ *\ err\ *\ X_{input}$
5: $end\ for$

b) Variant of adaptive filtering kernel explicitly in C

1: $for(k = k_0 + filter\text{–}order; k < k_0 + N + filter\text{–}order; k++)$
2: $\quad \{$
3: $\quad for(i = 0; i < filter\text{–}order; i++)$
4: $\quad\quad Y_{estimated}[k] = Y_{estimated}[k] + W[i] * X_{input}[i + k - filter\text{–}order]$
5: $\quad for(i = 0; i < filter\text{–}order; i++)$
6: $\quad\quad err[i] = Y_{estimated}[i + k - filter\text{–}order] - Y_{wanted}[i]$
7: $\quad for(i = 0; i < filter\text{–}order; i++)$
8: $\quad\quad W[i] = W[i] + \mu m\ *\ err[i]\ *\ X_{input}[i + k - filter\text{–}order]$
9: $\quad \}$

It was chosen just to illustrate the effect of the transformation IR^2 when we deal with the loop-carried dependencies within nested loops. There are no restrictions about the code. Note that we can process all kinds of algorithms even if they contain highly complex structures.

Given that k0 has been set to 4 (k0 = 4) and filter-order set to 3, we obtain the two outputs Yestimated[4]

Figure 2. Partial *Map* generated by processing Algorithm 1 in the proposed compiler.

and Yestimated[5] and their DFGs in Figure 2. We can now globally assess the parallel potential of the target program. Accordingly to Amdhal's law, the theoretical maximum achievable speedup Smax is limited by the inherent sequential fraction: "*fs*" and given as:

$$S_{\max} \simeq 1/f_s, \tag{1}$$

where f_s can be expressed as follows:

$$f_s = \frac{cost(workload\ enchained\ by\ true\ dependence)}{cost(total\ workload)} \tag{2}$$

The *Map* expresses the outputs program by their total enchained *defs*. Nodes that are aligned orthogonally form a depth level in the DFG from left to right as indicated in Figure 2.
Thus, Speedup may be evaluated as follows:

$$S_{max} \simeq \frac{total\ workload}{Number\ of\ depth\ levels}. \tag{3}$$

3. Instrumentation in the CHStone benchmark

It is important to know how much parallelism there is in the applications that are considered for parallelization. Unfortunately, to our knowledge, this issue has almost never been considered before in any parallelization tool. In other words, it has not been inspected concretely. The present parallelization framework acts as well as an instrument allowing the assessment of the intrinsic parallel potential and the theoretical speedups. Quantifying the intrinsic parallel potential accurately helps to deal with the resources management more rigorously, avoids overallocations, expands the parallel benchmarking options, and more importantly, it may assist to drive the adequate decisions in some parallelization scenarios. False dependences are still scrambled in the Map. False dependence expresses just the use of the memory location. It does not penalize the parallelization if data were privatized or renamed. So, the measures carried out here indicate the maximum achievable speedups for SMT in many/multicore systems if privatization is done properly in conjunction with an adequate mapping.

Table 1. Expectations of the maximum achievable speedups in a subset of CHStone benchmark.

Program	ADPCM	AES	GSM	BLOWFISH
Number of functions	15	11	12	6
Test vector length	100	16	160	120
Workload order	$\Theta(35500)$	$\Theta(6500)$	$\Theta(32300)$	$\Theta(79700)$
Cost(Dependence-enchained-workload)	256	206	440	2245
Speedup	138×	31×	73×	35×

3.1. CHStone benchmark

CHStone benchmark [28] consists of 12 programs with self-contained test vectors. Programs are selected from various application domains such as arithmetic, media processing, security, and microprocessor. A large variety of benchmark tests have been proposed in the literature. They are designed and tuned continuously to assess specific capabilities of systems and frameworks. The NASA Advanced Supercomputing (NAS) Parallel Benchmarks (NPB) are a small set of programs designed to help evaluate the performance of parallel supercomputers. They are derived from computational fluid dynamics (CFD) applications and consist of five kernels and three pseudoapplications. All three pseudoapplications included in this set relate to linear systems solving. SPEC's benchmarks were developed to evaluate the performance and energy efficiency of the newest generation of computing systems. Large sets of tests in SPEC's benchmarks were devoted to a variety of computing branches including Cloud, CPU, graphics/workstation, and also the high performance computing (HPC). The older releases SPEC CPU2000 and SPEC HPC2002 were devoted respectively to CPU and HPC branches tests. The integer component of SPEC CPU2000 consists of a dozen of applications that include specifically among others the well-known applications: gzip, vpr for FPGA placement and routing, gcc, parse and twolf for place and route simulation. SPEC HPC2002 consists of the SPECCHEM 2002 benchmark which is based on a quantum chemistry application, SPECENV 2002 benchmark which is based on a weather research, and the SPECSEIS 2002 benchmark which represents an industrial application that performs time and depth migrations used to locate gas and oil deposits. We privileged the use of CHStone tests. Kernels included in CHStone benchmark have equilibrate sizes, they have a good reputation and are given explicitly in C with a large syntax diversity and extensive inside-merged dependencies. For the context of the present investigation, it may constitute an

acceptable choice for the validation proof. The programs included in the chosen set are AES, ADPCM, GSM, and BLOWFISH.

Table 2. Expectations of the maximum achievable speedups in a subset of GSM functions.

Program	GSM		
Function	Autocorrelation()	Reflection_ coefficients()	Quantization_ and_coding()
Input vector length	160	8(LARc)	8(LARc)
Workload order	$\Theta(23900)$	$\Theta(6300)$	$\Theta(1600)$
Cost(enchained workload)	351	86	41
Speedup	68×	73×	39×

Table 3. Maximum achievable speedups in a subset of AES functions.

Program	AES	
Function	KeySchedule()	MixColumn_AddRoundKey()
Input vector length	16	16
Workload order	$\Theta(3100)$	$\Theta(800)$
Cost(enchained workload)	202	16
Speedup	15.3×	50×

3.2. Results and discussions

The measures exposed in Table 1 indicate the approximate limits of the maximum achievable speedups. In the context of SMT for many/multicore systems, these limits could be reached only if the parallelization framework implements the optimal solution available in the mapping–privatization schemes offered by the IR^2 structure. In other words, they are the maximum achievable speedups without considering the penalties occasioned by the false dependencies. The speedups are calculated according to Eq. (3). The workload of each program is also evaluated by the instrumentation within the tool.

In Tables 2–4, the inherent parallel potential of the most significant functions in the kernels GSM, AES, and BLOWFISH are given as well. They are instrumented individually. It is known that in cryptography, kernels data are highly correlated, which restricts the parallel potential as we can see it particularly in KeySchedule() and BF_set_key() functions. These functions belong respectively to the AES and BLOWFISH kernels. In GSM and ADPCM kernels, data are also somewhat correlated; however, it was revealed by instrumentation that parallel potential in these programs is relatively good.

A number of mapping–privatization schemes become available in the structure IR^2 already generated. They are not discussed in this paper, but in these schemes, the parallelization will be made less hard because we have to worry about only the false dependences. In counterpart, in the most of the state-of-art compilers [9–11], where the conventional thread–iteration-loop mapping scheme is used, all kinds of dependences (true dependences and antidependences) still merged among threads, which makes parallelization difficult. The pairwise mapping scheme that we call thread–final-def is one of the simplest schemes we can apply here without difficulties. Without further details, the overall concept of this mapping is that we have to assign each tree (or

Table 4. Maximum achievable speedups in BLOWFISH BF_set_key() function

Program	BLOWFISH
Function	BF_set_key()
Input vector length	18
Workload order	$O(11700)$
Cost(enchained workload)	350
Speedup	33×

graph) from the *Map* to one thread. In conjunction, we should apply data privatization. In these novel schemes of mapping, privatization is safe to make.

4. Conclusion

Most of today's parallelization compilers and semiautomatic tools usually implement the thread–iteration-loop mapping scheme. They have to recognize either the loops that are occasionally easy to schedule or the loops that can be slightly modified to fit in the thread–iteration-loop mapping scheme. Consequently, not all loops and not all regions are expected to be treated even if sometimes they hold a great parallel potential. In such autoparallelization frameworks, since they usually preserve the loops constructs and semantics, substantial dependencies of all kinds (true and false dependences) are expected to still keep among the loops iterations in the annotated/modified code. So, in order to beneficially apply this conventional thread–iteration-loop mapping scheme, one has to deal with the threads speculations, threads synchronizations, threads squashes, the questions of inter-threads results committing and/or the policies of the data privatization properly. All these concerns considerably constrain the parallelization task and make it difficult. We have proposed the incremental conditions committing approach. In the IR^2 representation, we have to worry about only the false dependences, which enables us to apply privatization with less difficulties. In this structure, several mapping schemes are offered and not just the conventional thread–iteration-loop scheme. We would be able to choose the most appropriate one to use with a safe privatization. We also have the choice to apply the proposed approach selectively on code portions: either on separate loops regions, on code segments, or also individually to functions.

References

[1] Li J, Sun J, Song Y, Zhao J. Accelerating MRI reconstruction via three-dimensional dual-dictionary learning using CUDA. J Supercomput 2015; 71: 2381-2396.

[2] Glowacz A, Pietron M. Implementation of digital watermarking algorithms in parallel hardware accelerators. Int J Parallel Prog 2017; 45: 1108-1127.

[3] Hidalgo-Pniagua A, Vega-Rodriguez MA, Pavon N, Ferruz J. A comparative study of parallel RANSAC implementation in 3D space. Int J Parallel Prog 2015; 43: 703-720.

[4] Okuyan E, Güdükbay U. Direct volume rendering of unstructured tetrahedral meshes using CUDA and OpenMP. J Supercomput 2014; 67: 324-344.

[5] Dagum L, Menon R. OpenMP: An industry standard API for shared-memory programming. IEEE Comput Sci Eng 1998; 5: 46-55.

[6] Ayguade E, Copty N, Duran A, Hoeflinger J, Lin Y, Massaioli F, Teruel X, Unnikrishnan P, Zhang G. The design of OpenMP tasks. IEEE Trans Parallel Distrib Syst 2009; 20: 404-418.

[7] Wang CK, Chen PS. Automatic scoping of task clauses for the OpenMP tasking model. J Supercomput 2015; 71: 808-823.

[8] Gonçalves R, Amaris M, Okada T, Bruel P, Goldman A. OpenMP is not as easy as it appears. In: IEEE 2016 System Sciences 49th Hawaii International Conference; 5–8 Jan 2016; Koloa, HI, USA. New York, NY, USA: IEEE. pp. 5742-5751.

[9] Blume W, Doallo R, Eigenmann R, Grout J, Hoeflinger J, Lawrence T, Lee J, Padua D, Paek Y, Pottenger B et al. Parallel programming with Polaris. Computer 1996; 29: 78-82.

[10] Bae H, Mustafa D, Lee JW, Aurangzeb, Lin H, Dave C, Eigenmann R, Midkiff SP. The Cetus source-to-source compiler infrastructure: overview and evaluation. Int J Parallel Prog 2013; 41: 753-767.

[11] Campanoni S, Jones TM, Holloway G, Wei GY, Brooks D. Helix: making the extraction of thread-level parallelism mainstream. IEEE Micro 2012; 32: 8-18.

[12] Liao C, Quinlan D, Panas T, de Supinski BR. A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries. In: International Workshop on OpenMP (IWOMP); 14–16 June 2010; Tsukuba, Japan. Heidelberg, Berlin: Springer. pp. 15-28.

[13] Zhang X, Navabi A, Jagannathan S. Alchemist: a transparent dependence distance profiling infrastructure. In: IEEE/ACM 2009 the 7th annual International Symposium on Code Generation and Optimization; 22–25 March 2009; Seattle, WA, USA. New York, NY, USA: IEEE. pp. 47-58.

[14] Chen T, Lin J, Dai X, Hsu WC, Yew PC. Data dependence profiling for speculative optimizations. In: International Conference on Compiler Construction; 29 March–2 April 2004; Barcelona, Spain. Heidelberg, Berlin: Springer. pp. 57-72.

[15] Kim M, Kim H, Luk CK. SD3: A scalable approach to dynamic data-dependence profiling. In: IEEE/ACM 2010 43rd Annual International symposium on micro-architecture; 4–8 December 2010; Atlanta, GA, USA. New York, NY, USA: IEEE. pp. 535-546.

[16] Li Z, Jannesari A, Wolf F. An efficient data-dependence profiler for sequential and parallel programs. In: IEEE 2015 International Parallel and Distributed Processing Symposium; 25–29 May 2015; Hyderabad, India. New York, NY, USA: IEEE. pp. 484-493.

[17] Sato Y, Inoguchi Y, Nakamura T. Whole program data dependence profiling to unveil parallel regions in the dynamic execution. In: IEEE 2012 International Symposium on Workload Characterization; 4–6 November 2012; La Jolla, CA, USA. New York, NY, USA: IEEE. pp. 69-80.

[18] Tian C, Feng M, Nagarajan V. Gupta R. Speculative parallelization of sequential loops on multicores. Int J Parallel Prog 2009; 37: 508-535.

[19] Campanoni S, Jones TM, Holloway G, JanapaReddi V, Wei GY, Brooks D. Helix: automatic parallelization of irregular programs for chip multiprocessing. In: ACM 2012 Proceedings of the tenth international symposium on code generation and optimization; 31 March–4 April 2012; San Jose, California, USA. New York, NY, USA: ACM. pp. 84-93.

[20] Johnson NP, Kim H, Prabhu P, Zaks A, August DI. Speculative separation for privatization and reductions. In: ACM 2012 Proceedings of the 33rd ACM SIGPLAN Conference on Programming Languages Design and Implementation; 11–16 June 2012; Beijing, China. New York, NY, USA: ACM. pp. 359-370.

[21] Tu P, Padua D. Automatic array privatization. In: International Workshop on Languages and Compilers for Parallel Computing; 12–14 August 1993; Oregon, USA. Heidelberg, Berlin: Springer. pp. 500-521.

[22] Li Z. Array privatization for parallel execution of loops. In: ACM 1992 Proceedings of the 6th International Conference on Supercomputing; 19–24 July 1992; Washington D. C., USA. New York, NY, USA: ACM. pp. 313-322.

[23] Li M, Zhao Y, Tao Y. Dynamically spawning speculative threads to improve speculative path execution. In: International Conference on Algorithms and Architecture for Parallel Processing; 24–27 August 2014; Dalian, China. Heidelberg, Berlin: Springer. pp. 192-206.

[24] Amini M, Creusillet B, Even S, Keryell R, Goubier O, Guelton S, Mcmahon JO, Pasquier FX, Péan G, Villalon P. Par4All: from convex array regions to heterogeneous computing. In: IMPACT 2012 2nd International workshop on polyhedral compilation techniques; Jan 2012; Paris, France.

[25] Blume W, Eigenmann R, Faigin K, Grout J, Hoeflinger J, Padua D, Petersen P, Pottenger W, Rauchwerger L, Tu P et al. Polaris: Improving the effectiveness of parallelizing compilers. In: International workshop on languages and compilers for parallel computing; 8–10 August 1994; Ithaca, NY, USA. Heidelberg, Berlin: Springer. pp. 141-154.

[26] Dave C, Bae H, Min SJ, Lee S, Eigenmann R, Midkiff S. Cetus: a source-to-source compiler infrastructure for multicores. Computer 2009; 42: 36-42.

[27] Psarris K, Klappholz D, Kong X. On the accuracy of the Banerjee test. J Parallel Distrib Comput 1991; 12: 152-157.

[28] Hara Y, Tomiyama H, Honda S, Takada H, Ishii H. CHStone: A benchmark program suite for practical C-based high-level synthesis. In: IEEE 2008 International Symposium on Circuits and Systems; 18–21 May 2008; Seattle, WA, USA. New York, NY, USA: IEEE. pp. 1192-1195.