# Path list traversal: a new class of SIMT flow tracking mechanisms

Caroline Collange, Nicolas Brunie


# Path list traversal: a new class of SIMT flow tracking mechanisms

**Caroline Collange and Nicolas Brunie**

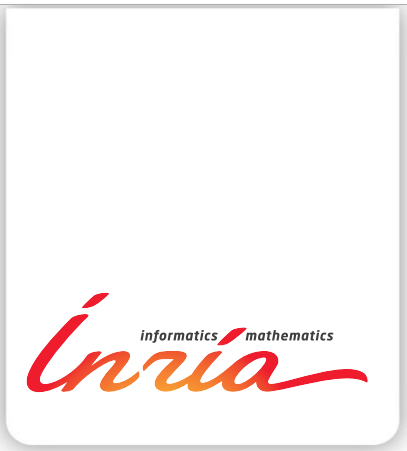
**RESEARCH REPORT**

**N° 9073**

May 2017

Project-Team PACAP

ISSN 0249-6399 ISRN INRIA/RR--9073--FR+ENG




# Path list traversal: a new class of SIMT flow tracking mechanisms

Caroline Collange* and Nicolas Brunie†

Project-Team PACAP

Research Report n° 9073 — May 2017 — 10 pages

**Abstract:** The SIMT execution model implemented in GPUs synchronizes groups of threads to run their common instructions on SIMD units. This model requires hardware or software mechanisms to keep track of control-flow divergence and convergence among threads. A new class of such algorithms is gaining popularity in the literature in the last few years. We present a new classification of these techniques based on their common characteristic, namely traversals of the control-flow graph based on lists of paths. We then compare the implementation cost on an FPGA of path lists and per-thread program counters within the Simty processor. The sorted list enables significantly better scaling starting from 8 threads per warp.

**Key-words:** GPU, SIMT, FPGA

* Inria
† Kalray

RESEARCH CENTRE
RENNES – BRETAGNE ATLANTIQUE

Campus universitaire de Beaulieu
35042 Rennes Cedex

# Parcours par liste de chemins: une nouvelle classe de mécanismes de suivi de flot SIMT

**Résumé :** Le modèle d'exécution SIMT employé dans les GPU synchronise l'exécution de groupes de threads afin d'exécuter leurs instructions communes sur des unités SIMD. Ce modèle nécessite des mécanismes matériels ou logiciels pour gérer la divergence et la reconvergence de contrôle entre threads. Une nouvelle classe de tels algorithmes émerge dans la littérature depuis quelques années. Nous présentons une classification de ces techniques sur la base de leur caractéristique commune, un parcours de graphe à base de liste. Nous comparons le coût de mise en œuvre sur FPGA de deux variantes du processeur Simty, l'une basée sur un tel mécanisme de reconvergence à base de liste triée et l'autre sur un mécanisme d'arbitrage entre compteurs de programme. La liste triée permet un passage à l'échelle significativement meilleur à partir de 8 threads par warp.

**Mots-clés :** GPU, SIMT, FPGA

# 1 Introduction

Graphics Processing Units (GPUs) are now established as general-purpose parallel processors, in particular for high-performance computing and machine learning. The success of GPUs can be attributed in a large part on their Single-Instruction Multi-Threading (SIMT) execution model. Following this model, a programmer writes Single Program, Multiple Data (SPMD) code as one program run by many threads. GPU hardware executes program instructions in Single Instruction Multiple Data (SIMD) units by grouping threads into *warps*, and running threads of each warp in lockstep so they execute the same instruction at the same time. GPU programming models are progressively joining CPU parallel programming models, both at the language level through CUDA C and C++ [12], OpenACC and OpenMP 4, and at the platform level through initiatives like HSA [14].

However, GPU instruction sets maintain a key difference with "traditional" CPU instruction sets. Indeed, GPU instruction sets need to convey the explicit structure of the control-flow graph to the microarchitecture. Unifying CPU and GPU instruction sets constitutes the next step on the road toward a common programming model. The key issue to create a unified instruction set consists in managing branch divergence and convergence as threads of a warp follow different control paths in the program, by only relying on conditional and indirect branches [2, 5, 6, 7].

We present Section 2 a survey of list-based methods that started to spread in the GPU literature in the last few years. We then describe Section 3 the sorted context table mechanism implemented in the Simty core, and compare its implementation cost with PC-based arbitration.

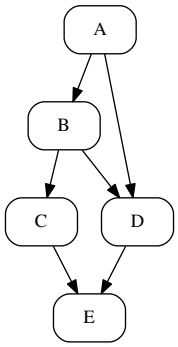
# 2 Managing thread divergence and convergence

Since our prior survey [2, 5], several novel SIMT flow tracking methods have been proposed in the literature or implemented in products. Among these, we focus on the class of list-based methods. We introduce a new taxonomy to characterize these techniques.

**Terminology** A *path* is characterized by a Program Counter (PC) and the subset of threads in a warp that have this PC. The execution context associated to a warp can be represented either by the vector of individual PCs of each thread of the warp, or by a complete disjoint list of path. By complete and disjoint, we mean that at any time, each thread of the warp belongs to one and only one path. During execution, the number of paths in the list may vary between one and the number of threads in a warp.

Following branches in SIMT may be seen as traversing the control-flow graph. The processor follows one path of each warp, that we name the *active* path. The other paths are saved in a list. Control divergence consists in splitting the active path into two new paths. One path is saved in the list, and execution continues on the other path. Convergence consists in merging the active path with a path of the list, or two paths of the list, when they have the same PC.

**Traversal orders** Different divergence and convergence management policies may be expressed as different graph traversal orders. Figure 1 compares several traversal orders obtained for code of the form `if(A && B) C; else D; E;`. The set of threads of each path is represented as a vector of bits, or *mask*: bit $i$ of the vector is set if and only if thread $i$ of the warp belongs to the set. We discuss each of the traversal orders in turn.




(a) Control-flow graph

| | | |
|---|---|---|
| A 1111 | | |
| B 1011 | (D 0100) | |
| C 1000 | (D 0011) | (D 0100) |
| D 0111 | (E 1000) | |
| E 1111 | | |

(b) Depth-first traversal

| | | |
|---|---|---|
| A 1111 | | |
| D 0100 | (B 1011) | [E 0000] |
| B 1011 | [E 0100] | |
| C 1000 | (D 0011) | [E 0100] |
| D 0011 | [E 1100] | |
| E 1111 | | |

(c) Traversal by minimal activity

| | |
|---|---|
| A 1111 | |
| B 1011 | (D 0100) |
| D 0111 | (C 1000) |
| C 1000 | (E 0111) |
| E 1111 | |

(d) Breadth-first traversal

| | |
|---|---|
| A 1111 | |
| B 1011 | (D 0100) |
| C 1000 | (D 0111) |
| D 0111 | (E 1000) |
| E 1111 | |

(e) Traversal by minimum PC

| | | |
|---|---|---|
| A 1111 | | |
| B 1011 | D 0100 | |
| C 1000 | D 0011 | (E 0100) |
| E 1111 | | |

(f) Parallel traversal

Figure 1: Examples of traversal orders showing the list state at each step. Thread 0 goes through graph blocks ABCE in order, thread 1 goes through ADE and thread 2 and 3 go through ABDE. Entries saved in the list are denoted with parentheses, and synchronization points with brackets.

## 2.1 Depth-first traversal

The simplest order consists in a depth-first graph traversal (Figure 1b). On structured control flow, depth-first traversal follows the nesting levels of control-flow. Convergences happens in the reverse order of divergences. Depth-first traversal is implemented by maintaining the path list as a stack.

Depth-first traversal leaves open the choice of which path should be followed first after a divergence. We may adapt to this means the stack-based scheduling technique we previously proposed [7, 5]. In order to favor convergence, divergent branches are handled in a different way following their direction, so the smallest PC is given priority. For a divergent forward branch (jump to a higher PC), the path of the branch target is pushed onto the stack, and execution continues on the next instruction. For a divergent backward branch, the path of the next instruction is pushed onto the stack, and execution continues at the branch target. On a uniform forward branch, the branch target is compared with the top of the stack PC. If the target is greater, the top of the stack is swapped with the active path (for instance from C to D on Figure 1b). When the PC is equal to the top of the stack PC, the latter is popped and merged with the active path to enable convergence (case of D). These steps ensure that the path with minimum PC is given the priority.

Although depth-first graph traversal involves a stack of addresses and masks like stack-based techniques from the Pixar Chap project [17] used is some contemporary GPUs, it is based on fundamentally different principles: the path stack represents here a list of future directions, rather than a global state.

## 2.2 Depth-first traversal with stack depth minimization

The AMD Graphics Core Next 2 architectures manages divergence mostly in software. It offers however some hardware-based divergence support for some irreducible control flow cases [1]. According to this scheme, the compiler delimits a single-entry, single-exit region using a *Fork* instruction at entry and a *Join* instruction at exit. At runtime, divergent branches within the region are handled the following way: (a) the path taken by the majority of threads is pushed onto the stack, and (b) execution continues on the path taken by the minority. When the *Join* instruction is reached, the top of the stacked is popped and execution is transferred to the corresponding path. Once the stack is empty, execution continues at the instruction following *Join* with the initial mask.

The choice to follow the path taken by the minority first guarantees that at each step, the active thread count is at least halved. The stack depth is thus bound by $\log_2(n)$ for $n$ threads per warp, versus $n$ for a generic list scheduling. This optimization is especially important in this context as the stack is implemented using registers, which are allocated statically regardless of actual branch divergence. On the other hand, the control flow subgraph is traversed as a tree: no partial convergence is possible until reaching the *Join* instruction. In the example of figure 1c, block D is thus traversed twice. Hence, this technique offers no convergence benefit for irreducible graphs, and excludes non-trivial loops.

## 2.3 Breadth-first traversal

Traversing the control-flow graph depth first may lead to deadlocks on programs that perform synchronization between threads. In particular, when one thread acquires a lock on a path while another thread on another path performs a busy waiting loop on the lock, depth-first traversal may keep scheduling the thread on the waiting loop, and the lock never gets released. Breadth-first traversal (Figure 1d) ensures fairness across paths and avoids such deadlocks [9]. It amounts to consider the path list as a queue. Nevertheless, breadth-first traversal still demands synchronization at "safe" convergence points.

A patent application, which might describe the implementation of the NVIDIA Volta architecture, proposes a hybrid traversal order [8]. It traverses structured control-flow paths depth-first, then unstructured paths breadth-first.

## 2.4 Traversal by priority order

Priority-based traversal consists in sorting the path list according to an order based on the list contents. The DWF architectures implements a path list whose threads are remembered by their identifiers [11]. This representation generalizes masks by allowing a path to include threads from multiple warps[1]. The article introducing DWF considers several priority-based path scheduling policies, including thread count majority, minority, and minimal PC.

The Maven architecture considers a *1-stack* policy based on the minimum PC. A shift register implements a systolic sort of the path list [16]. This implementation is practical as the architecture considered handles one single warp. A variation named *2-stack* modifies the sorting order to handle loops with several back edges more efficiently. On the other hand, *2-stack* does not appear to support nested loops.

The choice of the minimum PC is accurate for threads executing the same function, but is unable to arbitrate between threads running different functions. We proposed giving priority to

[1] In our terminology, Warp designates a statically-set group of threads, unlike the convention adopted in article [11].

the maximal function call depth, then to the minimum PC in case of a tie [5]. This policy is implemented in Simty [6] and evaluated Section 3.

Sorting the path list by PC guarantees that convergence always happen between adjacent paths within the list. Hence, convergence may be detected solely by comparing the active path with the head of the list, without requiring an associative search.

Recent patents describe a similar sort by function call depth and by PC [4], and a hardware implementation for a list of two path sorted by PC [13].

### 2.5 Parallel traversal

Graph traversal provides parallelism that can be exploited. By traversing multiple active paths simultaneously or alternatively, we may hide latency or increase execution throughput. Some first steps in this direction have combined a classic stack of masks with a set of paths traversed in parallel [18, 19]. However, using a stateful mask stack limits either the potential for parallel execution, or the timeliness of convergence [15].

By contrast, a list of path allows toggling paths easily between active and passive state. Convergence remains challenging, in particular on unstructured control flow. For instance, block D is traversed twice on the example of Figure 1f.

The SBI architecture with constraints bases convergence on an interval of addresses: convergence is restricted to the paths whose PC is comprised within an interval set by a convergence instruction [3]. The reasoning behind interval-based convergence is that a given structured control-flow nest is generally compiled to a contiguous block in the binary code. The Multi-path architecture relies on a convergence table [10]. Convergence is limited to single-entry, single-exit regions, and the size of the convergence table is unbounded. The HARP architecture represents each path with the identifiers of threads it contains like DWF, while offering extra flexibility for parallel path execution [15]. Convergence is based on an associative lookup in a convergence barrier table.

## 3 Comparison between arbitration and sorted list

We evaluate the respective costs of PC-based arbitration [5] and path tables [3]. We have implemented both mechanisms in RTL within the Simty processor [6]. Simty is a configurable multi-warp SIMT processor running the RISC-V instruction set. Its 10-stage pipeline is illustrated Figure 2. We present in turn each implementation of SIMT control flow tracking.

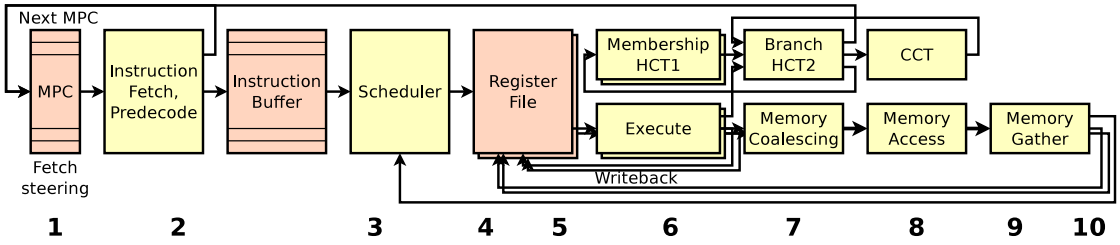



Figure 2: Overview of the Simty pipeline. Units dedicated to SIMT branching are detailed Figure 3.

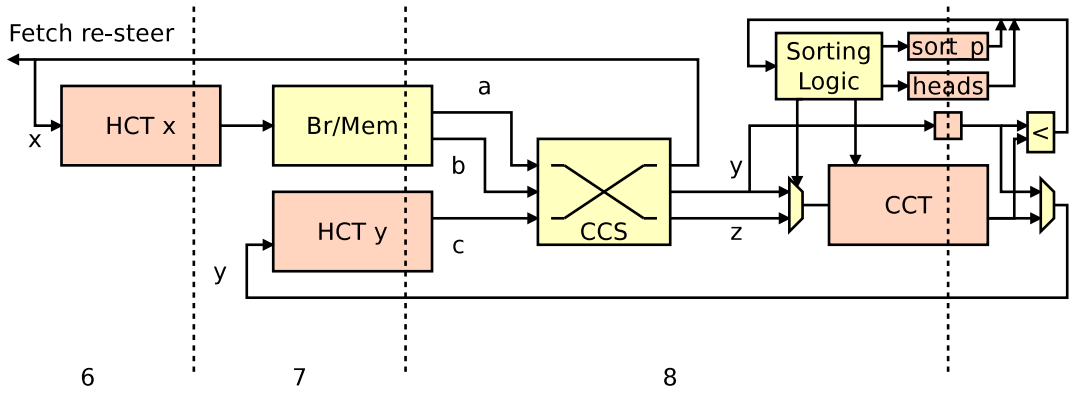



Figure 3: Detail of path tracking using path tables

## 3.1 Arbiter

The arbiter-based solution maintains one physical PC register per thread using a table indexed by the warp identifier. The table returns a vector made of the thread PCs for the warp. The maximal function call depth or minimal PC is selected using a parallel reduction based on a tree of comparators and multiplexers. For implementation simplicity reasons, we did not retain the unified arbiter option that we originally proposed, as memory arbitration and branch arbitration occur at different pipeline stages in Simty.

## 3.2 Sorted list

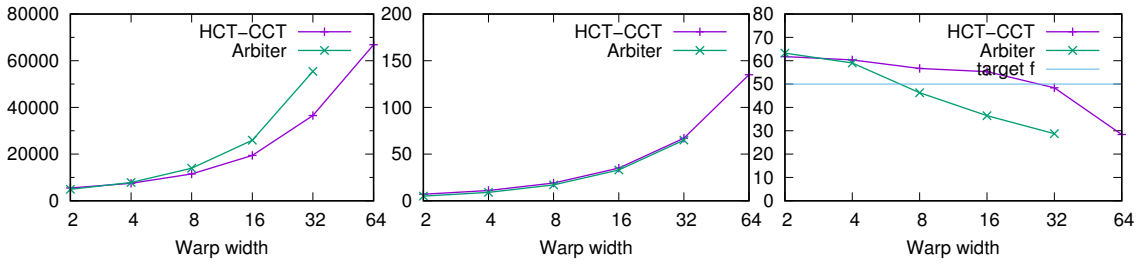
In the list-based version, the path tracking unit is pipelined in 3 stages (Figure 3). It is made of two "hot" path tables (Hot Context Table, HCT), one path compaction and sorting unit (Context Compact Sort, CCS), and one "cold" path table (Cold Context Table, CCT). The CCT is sized to support the worst case of one unique thread per path. From the active path, the branch/memory unit creates up to two new paths $a$ and $b$. The CCS unit sorts these paths together with the path $c$ coming from the second HCT, and merges paths that have the same PC and call depth, producing between 1 and 3 paths $x < y < z$. The fetch stage is resteered to the PC of path $x$, which is always valid. If all three paths are valid, $z$ is inserted in the CCT. If only path $x$ is valid, $y$ is popped from the CCT head. Paths $x$ and $y$ are then written back in the first and second HCT, respectively.

In order to enable single-cycle sorting, the CCS is based on a latency-optimized circuit rather than a sorting network. It consists of parallel comparators for order and equality between all pairs of paths, followed by three-input multiplexers driven by boolean functions of comparator results.

Maintaining two HCTs is not strictly necessary as far as functionality is concerned. However, the second HCT minimizes branch resolution time. Indeed, the second HCT path is ready to take over the active path as soon as needed. The CCT remains outside the critical path of the new PC calculation.

Whenever the CCT is idle (no insertion nor extraction), a state machine progressively sorts its contents. The state machine is temporally multiplexed between warps. Each entry is compared in turn with path $y$ of the second HCT. When entries are not in their expected order, they are swapped on the next cycle[2]

[2]If an insertion or extraction happens in the meantime, the swap operation is canceled to ensure the CCT remains self-consistent.




(a) Area (LEs) (b) Block RAM utilization (M9K) (c) Frequency (MHz)

Figure 4: Synthesis results after place and route of the Simty core, comparing HCT and CCT-based divergence tracking with arbiter-based divergence tracking, for 8 warps.

### 3.3 Evaluation

We compare post-place and route synthesis results on an Altera Cyclone IV EP4CE115 FPGA on Figure 4. The area is expressed in logic elements (LEs) and in 128 × 32-bit SRAM blocks (M9K). Target frequency is 50 MHz. Place and route of the arbiter-based version fails for 64 threads per warp, with a synthesis-time area estimate of about 91000 LEs.

Starting from 8 theads per warp, the overhead of PC arbitration becomes significant, especially in logic area. Indeed, beside the combinatorial logic of comparators, vectors of PCs require wide and shallow memories, which are implemented with individual flip-flops on this FPGA generation. On the other hand, the narrow memories of the HCTs and CCTs are well suited to FPGA resources.

The frequency estimate also drops below the target frequency of 50 MHz due to the high latency of arbitration. These results do not consider pipeline rebalancing: in practice, PC arbitration should be allotted one or more extra pipeline stages to preserve cycle time.

The area cost of the whole CCT control logic (managing insertions, extractions and sorting) ranges from 316 LEs for 2 threads/warp to 633 LEs for 64 threads/warp, corresponding to respective overheads of 6% and 1% for the whole core. The price to pay for sorting the CCT is thus acceptable, and significantly lower than the cost of a reduction tree for arbitration.

## 4 Conclusion

List-based SIMT divergence tracking mechanisms are a promising alternative to the mask stacks of SIMD architectures from the 1980-1990 which are employed on many modern GPUs. List-based techniques combine the benefits of mask stacks with those of arbitration between independent PCs. Like mask stacks, they offer low cost and low latency. Like independent PC arbitration, they maintain compatibility with general-purpose instruction sets and support arbitrary control-flow, including exceptions and interruptions.

## References

[1] AMD. *Southern Islands Series Instruction Set Architecture*, Aug 2012.

[2] Nicolas Brunie and Sylvain Collange. Reconvergence de contrôle implicite pour les architectures SIMT. *Revue des Sciences et Technologies de l'Information - Série TSI : Technique et Science Informatiques*, 32(2):153–178, February 2013.

[3] Nicolas Brunie, Sylvain Collange, and Gregory Diamos. Simultaneous Branch and Warp Interweaving for Sustained GPU Performance. In *39th Annual International Symposium on Computer Architecture (ISCA)*, pages 49 – 60, Portland, OR, United States, 2012.

[4] Lin Chen. Executing subroutines in a multi-threaded processing system. US Patent 9,229,721, jan 2016.

[5] Caroline Collange. Stack-less SIMT reconvergence at low cost. Technical report, HAL CCSD, September 2011.

[6] Caroline Collange. Simty: a synthesizable general-purpose SIMT processor. Research Report RR-8944, Inria Rennes Bretagne Atlantique, August 2016.

[7] Sylvain Collange, Marc Daumas, David Defour, and David Parello. Étude comparée et simulation d'algorithmes de branchements pour le GPGPU. In *SYMPosium en Architectures nouvelles de machines (SYMPA)*, 2009.

[8] Gregory Frederick Diamos, Richard Craig Johnson, Vinod Grover, Olivier Giroux, Jack H Choquette, Michael Alan Fetterman, Ajay S Tirumala, Peter Nelson, and Ronny Meir Krashinsky. Execution of divergent threads using a convergence barrier. US Patent App. 14/798,265, jul 2015.

[9] Ahmed ElTantawy and Tor M Aamodt. MIMD synchronization on SIMT architectures. In *49th Annual IEEE/ACM International Symposium on Microarchitecture*, 2016.

[10] Ahmed ElTantawy, Jessica Wenjie Ma, Mike O'Connor, and Tor M Aamodt. A scalable multi-path microarchitecture for efficient GPU control flow. In *International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[11] Wilson WL Fung, Ivan Sham, George Yuan, and Tor M Aamodt. Dynamic warp formation: Efficient MIMD control flow on SIMD graphics hardware. *ACM Transactions on Architecture and Code Optimization (TACO)*, 6(2):7, 2009.

[12] Mark Harris. *CUDA 9 Features Revealed: Volta, Cooperative Groups and More.* NVIDIA Parallel ForAll, may 2017.

[13] Rune Holm and David Hennah Mansell. Scheduling program instructions with a runner-up execution position. US Patent 9,436,473, sep 2016.

[14] Wen-mei W Hwu. *Heterogeneous System Architecture: A new compute platform infrastructure.* Morgan Kaufmann, 2015.

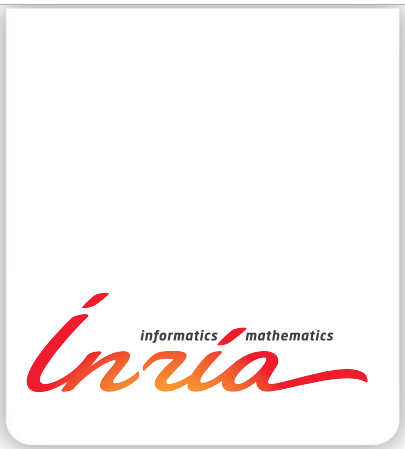
[15] Ahmad Lashgar, Ahmad Khonsari, and Amirali Baniasadi. HARP: Harnessing inactive threads in many-core processors. *ACM TECS*, 13(3s), 2014.

[16] Yunsup Lee, Rimas Avizienis, Alex Bishara, Richard Xia, Derek Lockhart, Christopher Batten, and Krste Asanović. Exploring the tradeoffs between programmability and efficiency in data-parallel accelerators. In *ACM SIGARCH Computer Architecture News*, volume 39, pages 129–140. ACM, 2011.

[17] Adam Levinthal and Thomas Porter. Chap - a SIMD graphics processor. In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '84, pages 77–82, 1984.

[18] Jiayuan Meng, David Tarjan, and Kevin Skadron. Dynamic warp subdivision for integrated branch and memory divergence tolerance. *SIGARCH Comput. Archit. News*, 38(3):235–246, 2010.

[19] Minsoo Rhu and Mattan Erez. The dual-path execution model for efficient GPU control flow. In *International Symposium on High Performance Computer Architecture (HPCA2013)*, pages 591–602. IEEE, 2013.




**RESEARCH CENTRE**
**RENNES – BRETAGNE ATLANTIQUE**

Campus universitaire de Beaulieu
35042 Rennes Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399