Activity Counter: New Optimization for the Dynamic Scheduling of SIMD Control Flow

Ronan Keryell *
Centre de Recherche en Informatique
École des Mines de Paris
77305 FONTAINEBLEAU Cedex, FRANCE
keryell@cri.ensmp.fr

Nicolas Paris *
Hyperparallel Technologies
École Polytechnique X-POLE
91128 PALAISEAU Cedex, FRANCE
paris@hyperparallel.polytechnique.fr

Abstract

SIMD or vector computers and collection-oriented languages, like C*, are designed to perform the same computation on each data item or on just a subset of the data. Subsets of processors or data items are implemented via an *activity* bit and a stack of activity bits when subsets of subsets are supported. This method is also used in VLIW processors through *if-conversion* to implement parallel control flow as in SIMD computers. We present a new method of dynamic sheduling of several SIMD control flow constructions which can be nested. Our implementation of activity stacks is based on *activity counters*. At a given stack depth n, the number of memory bits required is $\log_2 n$, whereas previous implementations require n bits. The local controller is of equivalent complexity in both cases. This algorithm is useful for SIMD, vector or VLIW machines and for compilers of collection-oriented languages on MIMD computers.

1 Introduction

The data-parallel programming model is seen as an acceptable solution to efficiently program many parallel applications on massively parallel machines. In this model, a single program is applied on different instances of data, spread across different processors, to gain use of parallelism on SIMD or MIMD machines.

In an SIMD computer there is a unique instruction flow and thus performs the same operation on different data. But a lot of numerical problems, like solving partial differential equation problems, often need to apply different at the boundary conditions which are different from the ones used on the interior points. Such a control flow is often introduced from a sequential program through vectorization and *if-conversion* [1].

A similar problem arises in data-parallel collection-oriented languages like MPL, C* or POMPC [10] where an SIMD-like control flow must be managed even through function or procedure boundaries not known at compile time or with recursion. So a dynamic SIMD control flow is needed to follow the locked-step SIMD semantics.

The goal of these parallel approaches is to obtain maximal performance on straight regular data parallel problems. However, it is at least as important to deal correctly with data parallel control flow and its flexibility.

There seems to be an intrinsic contradiction in the commonly used SIMD control flow model and the need for a local instruction stream, *i.e.* a bounded dissynchronization in the synchronous SIMD model, to deal with parallel control flow. This contradiction is resolved in turning off some processor elements (PEs) according to local conditions in SIMD machines, the *activity*. The nestling of several parallel `if` is usually managed with an activity stack but here we present an optimization of this method with an *activity counter* instead of a stack.

Section 2 presents our new algorithm with some examples applied to POMPC parallel control flow operators. Section 3 compares the activity stack with our method according to time and space complexity, for SIMD and MIMD, hardware and software. Section 4 presents related work.

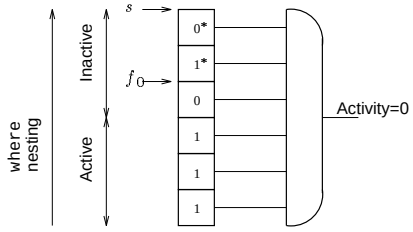

Figure 1: Example of a mask stack.

2 Activity counter

If we carefully look at the activity bit stack, we see it is only used to determine the level of inactivity. Figure 1 shows an example of a nest of 6 parallel control flow statements, where the first three ones have `true` conditions (shown as "1" in the figure) and the condition is `false` after the third one (represented by "0").

Before the first `false` condition, the stack only contains 1s, indicating that the PE is executing the code. The exit of a conditional block does not change this activity:

*Major parts of this work were made when the authors were with the Laboratoire d'Informatique de l'École Normale Supérieure, 45 Rue d'ULM, 75005 PARIS, FRANCE. This research and the POMP project were partially funded by the French Research and Technology Ministry, Thomson Digital Image, the CNRS (National Center of Scientific Research), the LIENS, the École Normale Supérieure, the PRC-ANM.

Table 2: Semantic of the `push` and `pop` operations on the activity counter.

Operation	Precondition	Action
`push`(*cond*)	$c \neq 0$	$c \leftarrow c + 1$
	$(c = 0) \wedge (cond = 0)$	$c \leftarrow 1$
	$(c = 0) \wedge (cond = 1)$	$c \leftarrow 0$
`pop`	$c \neq 0$	$c \leftarrow c - 1$
	$c = 0$	$c \leftarrow 0$

the PE remains active. These 1s do not have any intrinsic significance in the stack.

When a PE reaches a local `false` condition, it becomes inactive for all its included blocks. The current activity is the logical *and* of the history of activity, *i.e.* all the activity bits on the stack. Once a 0 bit is pushed on the stack, all the following bits on the stack no longer have meaning (represented with a "*" in Figure 1) since the activity is 0 (inactive).

2.1 Factorization

Indeed the only useful information in this stack is the nestling level of parallel conditional blocks after the first idle block, which indicates when a PE can resume execution. Therefore, it seems a waste of hardware to use a stack where a plain counter should be enough.

Let `push`(*cond*) and `pop` be the two operations controlling the stack $(a_i)_{i \in \mathbb{N}}$. We can analyze their functionality according to f_0, the rank of the first 0 on the stack, and s the current size of the stack, according to Figure 1. The activity of a PE is defined by $\mathcal{A} = \bigwedge_{i=0}^{s-1} a_i$. The PE is active if $\mathcal{A} = 1$ and idle if $\mathcal{A} = 0$.

By definition, PEs are all active at initialization time, so $s = 1$, $a_0 = 1$ (active), $f_0 = s + 1$ when there is no 0 in any stack element. For simplicity a `pop` on an empty stack returns an activity `true`.

Table 1 gives an operational semantics of the activity stack. A PE is active if and only if $f_0 = s + 1$, when there is no 0 in the stack. In fact, it is more interesting to do the variable exchange $c = s+1-f_0$ because only a comparison to 0 is necessary. This form is easier to implement in hardware and often even in software [7, 8]. The basic manipulations on c are the same as on f_0: increment or decrement, load or store, as shown on Table 2.

The `push`(*cond*) when $c = 0$ can be simplified to $c \leftarrow \neg cond$. A more detailed proof of the equivalence between an activity stack and an activity counter for parallel control flow can be found in [3, 9].

2.2 Application to a data parallel language

Now we can use this mechanism to implement classical parallel control flow operators such as those in the POMPC C-based language [10]. We present only the `where` and the `switchwhere` but the method is also used for the `whilesomewhere`, the `return` of a parallel function or procedure.

Table 3: Implementation of the `where`/`elsewhere` with an activity counter.

Operation	Precondition		Action
`where`(*cond*)	$c \neq 0$	*(idle)*	$c \leftarrow c + 1$
	$c = 0$	*(active)*	$c \leftarrow \neg cond$
`elsewhere`	$c \leq 1$	*(activatable)*	$c \leftarrow \neg c$
	$c \not\leq 1$		$c \leftarrow c$
End of the `where` `/elsewhere`	$c \neq 0$	*(idle)*	$c \leftarrow c - 1$
	$c = 0$	*(active)*	$c \leftarrow 0$

2.2.1 `where`

The basic operator is the `where`/`elsewhere` pair which is found in most data parallel languages from FORTRAN 90 to C*.

The `where` is equivalent to the `push` operator but we have to translate the `elsewhere`. A PE is active in an `elsewhere` if and only if the PE was inactive due to the *last* `where`, *i.e.* the inactivity level $c = 1$. The value 1 can be seen here as a special value that codes for an "activatable" state for the `where` or `elsewhere` block.

An implementation is presented in Table 3.

2.2.2 `switchwhere`

The compilation of a `switchwhere`, the parallel extension of the language C `switch`, also has several states. A PE can be:

1. inactive before the `switchwhere`;
2. active in a `case` (after matching a value) or in a `default`;
3. inactive in a `case`, waiting for a matching value;
4. inactive in the `switchwhere` because of a break, until the `switchwhere` exit.

The `break` is similar to the `whilesomewhere` one. An example of state coding we use is $c = 1$ for the state 3 and $c = 2$ for the state 4, as shown in Table 4.

3 Activity counters versus activity stacks

3.1 On an SIMD machine

The counter method needs a counter with $\log_2 c$ bits per PE if at most c levels of parallel conditional blocks are nested. If each PE has an L-bit operator, a PE needs $\lceil \frac{1}{L} \log_2 c \rceil$ cycles of duration t to do an activity counter operation.

The activity stack needs only 1-bit manipulation on each PE and takes a time t, but needs a stack pointer to manage

Table 1: Semantics of the `push` and `pop` operations on the activity stack.

Operation	Behavior	Precondition	Action
`push`(*cond*)	$s \leftarrow s+1$	$f_0 \neq s+1$	$f_0 \leftarrow f_0$
	$a_s \leftarrow$ *cond*	$(f_0 = s+1) \wedge (\mathit{cond} = 0)$	$f_0 \leftarrow s$
		$(f_0 = s+1) \wedge (\mathit{cond} = 1)$	$f_0 \leftarrow s+1$
`pop`	*if*[a] $(s > 1), s \leftarrow s-1$	$f_0 \neq s+1$	$f_0 \leftarrow f_0$
	`return`(a_s)	$f_0 = s+1$	$f_0 \leftarrow s+1$

[a] Note that if the program is correct, this condition is always true.

Table 4: Implementation of the `switchwhere` with an activity counter.

Operation	Precondition		Action
`switchwhere`(*value*)	$c \neq 0$	*(idle)*	$c \leftarrow c+2$
	$c = 0$	*(active)*	$c \leftarrow 1$
`case` *constant* `:`	$(c = 1) \wedge$ (*value* = `constant`)		$c \leftarrow 0$
`break`	$c = 0$	*(active)*	$c \leftarrow 2^a$
`default :`	$c = 1$	*(activatable)*	$c \leftarrow 0$
`switchwhere` *closing*	$c \leq 1$		$c \leftarrow 0$
	$c \not\leq 1$		$c \leftarrow c-2$

[a] Must be relative to the current `switchwhere` block, if the `break` is included in one or more `where`/`elsewhere`.

the stack. Since the execution is SIMD, all the stacks are synchronous and the stack pointer can be:

- centralized on the scalar processor which broadcasts its value to the PEs;
- distributed with local pointers which evolve synchronously.

In the first case, it takes a time T on the scalar processor and the time is negligible on the PEs. In the second case, a time $t\lceil \frac{1}{L} \log_2 c \rceil$ is needed to control the stack pointer on each PE. The hardware complexity is c for a stack of 1 bit elements in each case, plus $\lceil \frac{1}{L} \log_2 c \rceil$ bits for the global stack pointer in the first case and $N\lceil \frac{1}{L} \log_2 c \rceil$ bits for the local stack pointers in the second case, for a N-PE computer.

The complexity of the three previous methods are summarized up in Table 5.

If the computer has only fine grain PEs, typically $L = 1$ or 4 bits, it is more interesting to subcontract the computation to the scalar processor with a global stack pointer. Indeed, the scalar processor is often larger and more powerful, so the stack pointer computation only uses few cycles, and even the broadcast is often shorter than the $t\lceil \frac{1}{L} \log_2 c \rceil$ required to deal with a local stack pointer or activity counter by L-bit slices. Moreover, 1-bit PEs have the advantage that they easily access memory with 1-bit. This method is used on computers such as the CM-2 or the MP-1.

The activity counter algorithm is particularly interesting for coarse grain SIMD machines and could be interesting in the MP-2. This method is used in our POMP MC88100-based SIMD computer [4, 8]. These computers often have short cycle time and the local memory access is slow in comparison to the PE cycle time.

3.2 On an MIMD machine

The complexity of our method for an MIMD machine is the same as in table 5 except that since there is no scalar processor, it is not interesting to have a global activity stack pointer and thus only local pointers or activity counters are necessary.

As for the SIMD computers, the same conclusions arise according to the size of the PEs. Activity counters can avoid the 1-bit stack management, specially inefficient on the coarse grain PEs which are in most MIMD computers. Besides, the activity counter on each PE reduces to $\mathcal{O}(\log c)$ the hardware complexity to store the activity.

But unlike SIMD computers, it is not worth implementing the activity counter in hardware since local conditional jumps are used *in fine* to efficiently emulate the activity corresponding to the counter value.

4 Related work

Methods to change control dependance in data dependence statically deal more or less with activity.

In [1] a complete guard is used and in [6] a minimum number of guard is produced to control activity.

Table 5: Complexity of the activity counter and activity stack methods.

Parallel conditioning	Computing complexity scalar	Computing complexity parallel	Hardware complexity	# broadcast
Stack (global pointer)	T	t	$Nc + \lceil \log_2 c \rceil$	1
Stack (local pointers)	ϵ	$t(1 + \lceil \frac{1}{L} \log_2 c \rceil)$	$N(c + \lceil \log_2 c \rceil)$	0
Activity counters	ϵ	$t \lceil \frac{1}{L} \log_2 c \rceil$	$N \lceil \log_2 c \rceil$	0

In [5], all the control flow information is kept in an "Exit" variable similar to our activity counter used for complex statements like `switchwhere` or `whilesomewhere` with `break`, `case` or `return`..

But none of these methods deals with dynamic scheduling, necessary for recursion or any procedure calls.

A counter methods is also used in [2] for dynamic sheduling in a dataflow-like architecture but there is no support for recursion.

5 Conclusion

We have developed a new method to dynamically deal with nested parallel control flow and recursion for SIMD and MIMD computers, and compilers for languages with collection oriented data parallelism.

This technique allows a reduction to a straight logarithmic term of the size in bits of memory used to keep track of the PE history, more efficient on coarse grain parallel computers and VLIW processors.

The optimization is also interesting for compilers targeted to modern MIMD computers when the nested parallel control flow cannot be resolved at compile time. For example, if different collections are mixed, interprocedural analysis is not performed or not possible, or if complex sub-array selections cannot be determined. If the activity counter method can often be replaced by MIMD local control flow, for complex nested case it seems a better choice.

At last, it is a way to compile nested parallel flow control flow in a "flat" normal form as in F90 or HPF where such a nestling is not allowed.

The activity counters are used in the POMP computer and also in the POMPC compiler for CM-2, MP-1, iPSC/860 and ARMEN.

6 Acknowledgements

The authors of this paper would like to acknowledge many useful discussions with all the members of the POMP team since the beginning of the project.

Special thanks are due to Luc BOUGÉ and his team, especially Jean-Luc LEVAIRE, for their discussions on SIMD semantics in parallel control flow and for their interest for the domain and our work.

At last but not the least, the authors are indebted to Kathryn MACKINLEY, François IRIGOIN and Pierre JOUVELOT for their invaluable comments and their appropriate suggestions.

References

[1] J. R. ALLEN, Ken KENNEDY, Carrie PORTERFIELD, and Joe WARREN. << Conversion of Control Dependence to Data Dependence >>. In *Conference Record of the Tenth Annual ACM Symposium on Principles Of Programming Languages*, pages 177–189. Association for Computing Machinery, January 1983.

[2] Carl J. BECKMANN and Constantine D. POLYCHRONOPOULOS. << Microarchitecture Support for Dynamic Scheduling of Acyclic Task Graphs >>. In *The 25th Annual International Symposium on Microarchitecture*, volume 23(1-2), pages 140–148. ACM SIG MICRO Newsletter, December 1992.

[3] Luc BOUGÉ and Jean-Luc LEVAIRE. << Control structures for data-parallel SIMD languages: semantics and implementation >>. *Future Generation Computer Systems*, 8(3-4):363–378, 1992.

[4] Philippe HOOGVORST, Ronan KERYELL, Philippe MATHERAT, and Nicolas PARIS. << POMP or How to Design a Massively Parallel Machine with Small Developments >>. In *PARLE '91 Parallel Architectures and Languages Europe*, volume 505(I), pages 83–100. Lecture Notes in Computer Science, Springer-Verlag, June 1991. Available by `ftp anonymous` on `spi.ens.fr` in the file `pub/reports/liens/liens-91-5.A4.ps.Z`.

[5] Bor-Ming HSIEH, Michael HIND, and Ron CYTRON. << Loop Distribution with Multiple Exits >>. In *Supercomputing '92 (Proceedings)*, pages 204–213. The Institute of Electrical and Electronics Engineers, Inc., November 1992.

[6] Ken KENNEDY and Kathryn S. MCKINLEY. << Loop Distribution with Arbitrary Control Flow >>. In *Proceedings of Supercomputing '90*, pages 407–416. The Institute of Electrical and Electronics Engineers, Inc., November 1990.

[7] Ronan KERYELL. << POMP2 : D'un Petit Ordinateur Massivement Parallèle >>. Rapport de magistère, LIENS — Ecole Normale Supérieure, October 1989.

[8] Ronan KERYELL. << *POMP : d'un Petit Ordinateur Massivement Parallèle SIMD à Base de Processeurs RISC — Concepts, Etude et Réalisation* >>. PhD Thesis, Laboratoire d'Informatique de l'Ecole Normale Supérieure — Université Paris XI, October 1992.

[9] Jean-Luc LEVAIRE. << *Contribution à l'étude sémantique des langages à parallélisme de données; application à la compilation* >>. PhD Thesis, LIP — ENS Lyon, Université de Paris 7, February 1993.

[10] Nicolas PARIS. << Definition of POMPC (Version 1.99) >>. Technical Report LIENS-92-5-bis, Laboratoire d'Informatique de l'École Normale Supérieure, March 1992. Available by `ftp anonymous` on `spi.ens.fr` in the file `pub/reports/liens/liens-92-5-bis.A4.ps.Z`.