

# XPP-VC: A C Compiler with Temporal Partitioning for the PACT-XPP Architecture

João M. P. Cardoso and Markus Weinhardt

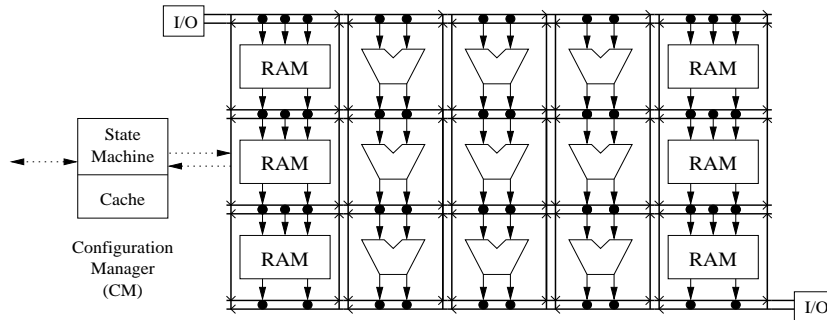
PACT Informationstechnologie AG, Muthmannstrasse 1, 80939 Munich, Germany  
jmpc@acm.org, mw@pactcorp.com

**Abstract.** *The eXtreme Processing Platform (XPP) is a unique reconfigurable computing architecture supported by a complete set of design tools. This paper presents the XPP Vectorizing C Compiler XPP-VC, the first high-level compiler for this architecture. It uses new mapping techniques, combined with efficient vectorization. A temporal partitioning phase guarantees the compilation of programs with unlimited complexity, provided that only the supported C subset is used. A new partitioning scheme permits to map large loops of any kind. It is not constrained by loop dependencies or nesting levels. To our knowledge, the compilation performance is unmatched by any other compiler for reconfigurable architectures. Preliminary evaluations show compilation times of only a few seconds from C code to configuration binaries and performance speedups over standard microprocessor implementations. The overall technology represents a significant step toward reconfigurable architectures which are faster and simpler to program.*

## 1 Introduction

Many applications are characterized by intensive data-stream processing and both low power consumption and high performance requirements. It is more and more evident that these requirements cannot be fully accomplished with today's microprocessor technology. Accelerating specific tasks using ASICs relieves some of the processing burden and adds some required features, but limits flexibility and requires high non-recurring engineering (NRE) costs and long design cycles. FPGAs eliminate the NRE costs and add flexibility, but still require hardware design expertise. Because of their fine-grain structure, FPGAs are not a particularly suitable target platform for compiling high level algorithmic descriptions.

New coarse-grained reconfigurable processors are being introduced to tackle these problems [1]. One of the new promising architectures is the XPP [2][3]. It features a coarse-grain, runtime reconfigurable ALU array. The XPP was designed to support pipelining, both instruction- and task-level parallelism, and dataflow and stream-based computations. The XPP-VC high-level compiler described in this paper was developed to drastically reduce the programming time of an XPP device, and to hide architecture details from the user. A novel temporal partitioning technique allows to map programs without size limits.



**Fig. 1.** Simplified structure of an XPP array.

This paper is organized as follows. The next section briefly describes the features of the XPP technology relevant for XPP-VC. Section 3 outlines compilation to the XPP and section 4 describes the temporal partitioning steps. Section 5 shows some experimental results, section 6 points out the main differences between this and previous work, and finally section 7 concludes the paper and enumerates ongoing and future work.

## 2 XPP Technology

The XPP technology consists of a reconfigurable computing architecture [3] delivered as a device or an IP core, and a complete development tool suite consisting of a place and router, a simulator, and a visualizer [2]. The tools use the proprietary *Native Mapping Language* (NML), a structural language with reconfiguration primitives [2].

The XPP architecture is based on a 2-D array of coarse-grained, adaptive processing elements (PEs), internal memories, and interconnection resources. XPP has some similarities with other coarse-grained reconfigurable architectures which were especially designed for stream-based applications (*e.g.*, KressArray [1]). XPP's main distinguishing features are its sophisticated configuration mechanisms and the data-flow protocols implemented in the interconnection resources.

**Array Structure:** Fig. 1 shows the structure of a simple XPP array. It contains a  $3 \times 3$  square of PEs in the center and one column of independent internal memories on each side. There are two I/O units which can either be configured as ports for streaming data or as interfaces for external RAM. The PEs perform common arithmetic and logical operations, comparisons, and special operations such as counters. During a configuration, each PE performs one dedicated operation. Each thick line in the figure represents several segmented busses which can be configured to connect the output of a PE with other PE's inputs. The array is attached to a *Configuration Manager* (CM) responsible for the runtime management of configurations, for loading configuration data from external memory and

writing it into the configurable resources of the array. Besides a state machine, the CM has cache memory for configuration data<sup>1</sup>.

**Data and Event Synchronization:** The interconnection resources of an XPP consist of two independent sets of busses: data and event busses. The number of busses is specific to a certain XPP. The data busses have a uniform bit width specific to the device type, while event busses carry one-bit control information. One value transmitted over a data or event bus is referred to as a data value or event, respectively. The busses differ considerably from conventional reconfigurable architectures (*e.g.*, FPGAs). They are not just wires to connect combinational logic or registers. There is also a ready/acknowledge protocol implemented in hardware which synchronizes the data values and events processed by the PEs. Additionally, pipelining registers can be switched on or off in the bus segments. Due to the synchronization, a PE operation or memory read is performed as soon as all necessary input values are available and the previous result has been consumed. Similarly, a value produced by a PE is forwarded as soon as all the busses connected to the PE's output have consumed the previous value. Thus it is possible to map a signal-flow graph directly to PEs, and to pipeline input data streams through it. The hardware synchronization protocols ensure that no values are lost, even in the case of pipeline stalls or during the configuration process. The clock-enable inputs of registers are not accessible to the programmer, and no explicit scheduling of operations is required. This simplifies application development considerably. Note that an XPP is a synchronous design operating at a fixed clock frequency.

Events transmit state information which can be used to control a PE's execution or memory accesses. There are also some special stream operations. MERGE uses an event to select one of two input data values, thereby merging two data streams, controlled by an event stream. MUX has a similar functionality, but discards the input data value not selected. Finally, DEMUX forwards its input data to the output selected by an event, and can be used for distributing a data stream into two. DEMUX can also be used to selectively discard data values from a stream. Thus conditional computations depending on the results of earlier PEs are feasible. Some PE operations generate events depending on results or exceptions, very similar to the state flags of a classical microprocessor. A counter, *e.g.*, generates a special event only after it has terminated. Events can even trigger a self-reconfiguration of the device as explained below.

**Configuration:** The XPP architecture is optimized for rapid and user transparent configuration. Every configurable object (*e.g.*, PE, memory) locally stores its current configuration state, *i.e.*, if it is part of a configuration or not (states "configured" or "free"). If a configuration is requested, the configuration data is first copied to the internal cache. Then the CM loads the configuration onto the

---

<sup>1</sup> It is possible to hierarchically combine several arrays with independent CMs which are then controlled by a master CM.

array, synchronizing with the configurable objects<sup>2</sup>. Once an object is configured, it changes its state to “configured”. This prevents the CM from reconfiguring an object which is still used by another configuration<sup>3</sup>.

Objects are released (*i.e.*, changed to state “free”) by events on a special input. This event is automatically broadcast along all connected PEs such that the entire configuration is removed. An event on the release input can be explicitly generated by the CM or created in the array itself by a PE. Hence running applications can request a self-reconfiguration of the device and thus it is possible to execute an application consisting of several phases without any external control. When a phase is finished, the PE detecting the end of the phase (*e.g.*, a counter) generates the event to be broadcast. Additionally, a PE can send an event to the CM to request the next configuration.

The configuration latency can be reduced by a prefetching mechanism: during the loading of a configuration onto the array other configurations can be copied to the CM cache. Thus it need not be copied from external memory when configurable objects become available. The same is true if a configuration was already loaded earlier and its configuration data is still in the CM cache.

Because of its course-grained nature, an XPP can be configured rapidly. Since only the configuration of those array objects actually used is necessary, the configuration time depends on the application.

### 3 Compiling C Code with XPP-VC

The XPP Vectorizing C Compiler XPP-VC is based on the SUIF compiler framework [4]. XPP-VC translates C programs to NML files. The currently supported C subset excludes `struct` and floating-point data types, pointers, irregular control flow (`break`, `continue`, `goto`), and recursive and operating system calls. A compiler options file specifies the parameters of the target XPP and the external memories connected to it. To access XPP I/O ports specific C functions are provided.

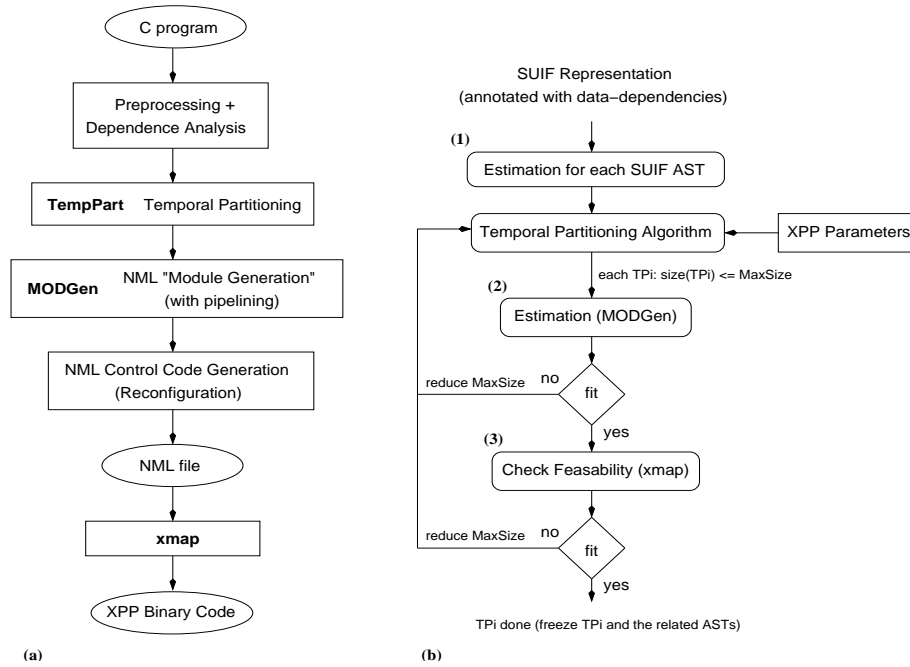
Fig. 2(a) shows the compilation flow. First, some architecture-independent preprocessing passes based on well-known compilation techniques [5] are executed. During this phase, user-annotated FOR loops are automatically unrolled. Then the compiler performs a data-dependence analysis and tries to vectorize inner program FOR loops. In XPP-VC, vectorization means that loop iterations are overlapped and executed in a pipelined, parallel fashion. This technique is based on the *Pipeline Vectorization* method [6]. Next, `TempPart` splits the C program into several temporal partitions if required (see section 4). Then, `MODGen` generates one NML module for each partition and NML reconfiguration code is added.

---

<sup>2</sup> However, differential configurations can also change an already “configured” object.

They only change parts of its configuration, *e.g.* a constant input. This feature is very useful for applications like adaptive filters.

<sup>3</sup> This method has another advantage if several arrays are combined. Then the individual CMs can configure their respective arrays independently [3]. No global synchronization is necessary.



**Fig. 2.** (a) XPP-VC compilation flow. (b) Temporal partitioning process realized by TempPart.

Finally, each NML module is placed and routed automatically by `xmap`, which uses an enhanced force-based placer with short runtimes. `xmap` also compiles the reconfiguration code.

We now describe how C code is transformed to NML code. First, program data is allocated on the XPP. By default, `MODGen` maps each program array to internal or external RAM, and scalar variables needing registers are stored in registers within the array. Next, a control/dataflow graph (CDFG), which directly reflects the NML structure needed, is generated. Straight-line code without array accesses is directly mapped to a dataflow graph. One PE is allocated for each operator in the CDFG. Because of the self-synchronization of operators on the XPP, no explicit control or scheduling is needed. The same is true for conditional execution. Both branches of an IF statement are executed in parallel and MUX operators select the correct output depending on the condition, as in many compilers targeting FPGAs, [7]. This data-driven execution of the operators automatically yields instruction-level parallelism. Fig. 3(a) shows a simple conditional statement and its CDFG.

In contrast, accesses to the same array have to be controlled explicitly to maintain the correct execution order. MERGE operators forward addresses and data in the correct order to the RAM, and DEMUX operators forward values read to the correct subsequent operator. State machines for generating the correct se-

quence of events (to control these operators) are synthesized by the compiler. I/O port accesses are handled in a similar way. Fig. 3(c) shows the implementation of the statements `x = arr[a]; y = arr[b];`. In this example, the events controlling the stream operators are simply generated by an event register with a feedback cycle. The register is preloaded with “0” and is configured to invert its output value.

IF statements containing array accesses or inner loops are not implemented with MUX operators as described above. Instead, DEMUX operators controlled by the IF condition to forward data only to the selected branch are used. The outputs of both branches are connected to the subsequent PEs. Since only one branch generates a value, no conflict occurs. With this implementation, only selected branches receive data and execute. Fig. 3(b) shows the conditional statement of Fig. 3(a) implemented this way.

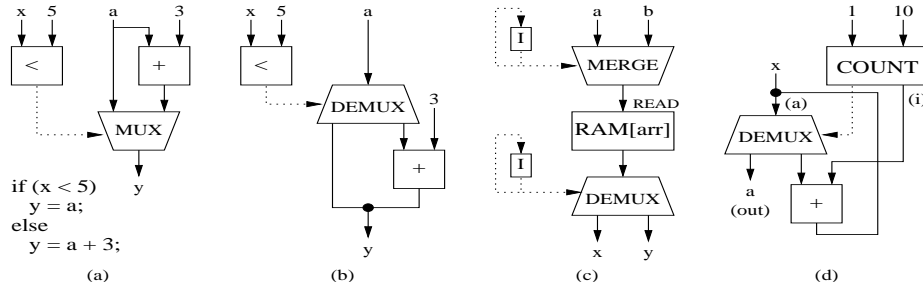
In loops, all variables updated in the loop body are treated as follows. The first iteration uses the input value of the variable, and the subsequent iterations use values generated in the previous iteration. In all but the last iteration, a DEMUX operator routes the outputs of the loop body back to the body inputs. Only the results of the last iteration are routed to the loop output. The events controlling the DEMUX are generated by the loop counter of FOR loops or by the comparator evaluating the exit condition of WHILE loops. Note that the internal operators’ outputs cannot just be connected to subsequent operators since they produce a result in each loop iteration. The required last value would be hidden by a stream of intermediate values. As opposed to registers in FPGAs, the ready/acknowledge protocol requires that values in XPP registers are read or explicitly discarded before a new value is accepted. Fig. 3(d) shows the (slightly simplified) CDFG for the following C code:

```
a = x;
for (i = 1; i <= 10; i++) a = a + i;
```

Note that the counter generates ten data values (1..10), but eleven events. After ten loop iterations a “1” event routes the final value of `a` to the output.

If the loop body contains array or I/O accesses, a loop iteration may only start after the previous iteration has terminated since the original access order must be maintained. The compiler generates events enforcing this. For generating more efficient XPP configurations, MODGen generates pipelined operator networks for inner program loops which have been annotated for vectorization by the pre-processing phase. In other words, subsequent loop iterations are started before previous iterations have finished. Data flows continuously through the operator pipelines. By applying pipeline balancing techniques, maximum throughput is achieved. Note that only IF statements implemented with MUX operators are currently pipelined. For many programs, additional performance gains are achieved by complete loop unrolling.

To reduce the number of array accesses, the compiler automatically removes some redundant array reads. When array references inside loops read subsequent element positions, the compiler only reads one element per iteration and generates shift registers to store the other values.



**Fig. 3.** C compilation examples. Data busses are represented as solid arrows, and event busses as dotted arrows. Black dots denote combined data busses: (b) and (d). I represents an event register with an inverted output (c).

## 4 Temporal Partitioning

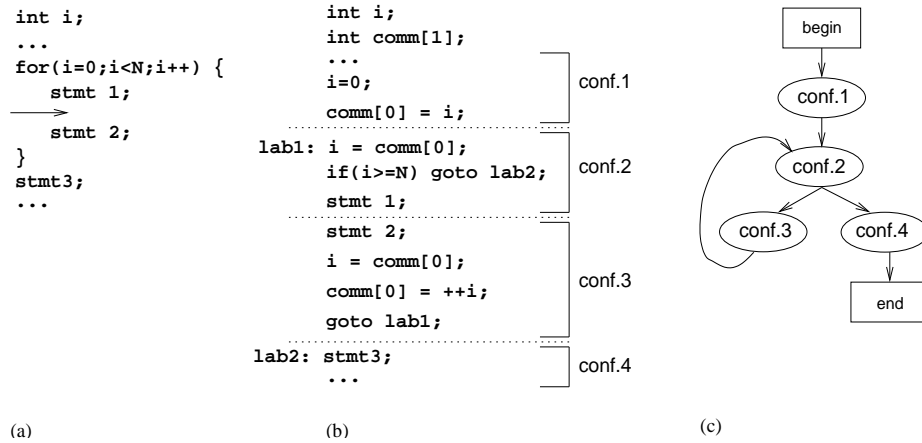
A program too large to fit in the target XPP can be handled by performing temporal partitioning, *i.e.*, by splitting it in several sequential parts such that each one is mappable. The problem we intend to solve can be formulated as follows. Given a C program and a specific XPP device, determine the set of feasible temporal partitions which results in the fastest overall program execution<sup>4</sup>.

Fig. 2(b) shows the main steps for the temporal partitioning process. A first level of estimations ((1) in Fig. 2(b)) computes the XPP resources needed for each SUIF abstract syntax tree (AST) in the code. Then, the temporal partitioning algorithm is performed. The algorithm starts with a partition for each AST in the highest level of the SUIF representation and then iteratively merges adjacent partitions. The first merges gives higher priority to the merging of non-loop ASTs since these merges can lead to performance gains. During the merging the algorithm checks if the resulting partitions are feasible, *i.e.*, leads to segments of code entirely mappable in the available XPP resources. The iterative process runs until a single partition is achieved, or no more merges are feasible. To decide on a certain merge the compiler considers that each new partition must define, on the control flow graph (CFG) of the program, regions of code with a single entry and possibly multiple exits.

Only if an AST, in the hierarchic level being traversed by the algorithm, does not fit into the available XPP resources, the algorithm is applied hierarchically to the inner representation of the AST. Partitioning of loops<sup>5</sup> is dealt with by a new

<sup>4</sup> Since in the XPP architecture the execution of each configuration needs three stages (fetching, configuring, and computation), we can reduce the global configuration overhead by finding the set of configurations that maximizes the overlapping of those stages considering also the use of prefetching. However, this issue is not the focus of this paper.

<sup>5</sup> Note that loops are fully executed in a single configuration whenever possible. This has the advantage of reusing an active configuration as long as possible.



**Fig. 4.** Applying the *loop dissevering* technique: (a) original code: the arrow identifies the place where the loop must be partitioned; (b) transformed code with identification of the configurations and with statements inserted to buffer the variable *i*; (c) graph representing the flow of configurations.

method, called *loop dissevering*<sup>6</sup>. The scheme transforms the loop into straight line code with a conditional jump to loop-exit or to the next iteration (see Fig. 4). The scheme uses the capability of a reconfiguration controller to orchestrate the flow of configurations needed (see Fig. 4(c)). In the case of the XPP, the controller is directly realized by the CM. In other architectures it can be supported by a host CPU. The transformation creates two additional configuration boundaries to preserve the initial functionality (see the configuration boundaries in Fig. 4(b) inserted before and after the loop). *Loop dissevering* can be applied to any type of loop and is not constrained by any type of dependences or nested structures (e.g., nested loops).

XPP implementation estimations derived from a high level SUIF representation are fast but not accurate. Since we need to ensure that each partition generated fits onto the available XPP resources, our methodology also uses two other estimation levels (see (2) and (3) in Fig. 2(b)). In level (2) each processed partition is checked with estimations performed using the CDFG generated by MODGen. If the size exceeds the available resources, the algorithm cancels the last merges, and proceeds tightening the size constraint (diminishing the maximum number of available resources). Level (3) tests if each configuration successfully checked in level (2) can be really mapped to the XPP. This checking uses functions of the placer and router (*xmap*). If the configuration does not fit, the algorithm once more cancels the last merges and proceeds tightening the size constraint.

<sup>6</sup> In some cases loop partitioning can be dealt with by applying *loop distribution* [5]. At the moment, however, *loop distribution* is not automatically applied.

Currently, arrays assigned to internal memories are used as inter-partition storage for scalar variables <sup>7</sup>, as illustrated by the communication of the variable `i` in Fig. 4(b) via the array `comm`. The assignment of the arrays (the initially used in the program plus the added ones to communicate data) to the XPP's internal memories is done based on their lifetimes. Those lifetimes are determined by the flow of configurations exposed. This permits, in some cases, to use fewer internal memories since they can be time shared among different configurations (*i.e.*, arrays in the same memory locations can use the same memory).

Finally, the XPP-VC compiler generates the configuration data for each partition and the code to program the CM which represents the flow of configurations.

## 5 Experimental Results

We have done a preliminary evaluation of the XPP-VC compiler with a set of benchmarks from different domains. Table 1 shows the main properties of the benchmarks used. They have from 18 to 228 lines of C code (LoC), from 2 to 16 loops, and from 2 to 6 array variables. The primary data size indicates the number of data values used for executing the benchmarks. Table 2 shows the results obtained with XPP-VC. The second column shows the CPU time (using a PC with a Pentium III @933MHz with 256MB of RAM under Linux) to compile each example with XPP-VC (from the C program to the binaries). In the experiments we use an XPP array of 16×16 PEs. However, in some of the cases with more than one partition we have reduced the maximum number of PEs used. Columns #Part, #PEs, and #Max represent the number of partitions, number of PEs used (in the cases with more than one partition, we show the number of PEs of the largest configuration and the sum of PEs for all configurations), and the maximum number of PEs executing per clock cycle (as an indication of ILP degree), respectively. Columns 5 and 6 show the overall computation time of the configurations on the array and the total execution time (taking into account setup, fetching, configuring, data communication and computation), respectively, both in number of clock cycles. The last column shows the speedup obtained with XPP @150MHz implementations (using the total execution time) over the examples compiled with `gcc` and running on a PIII @933MHz.

The compiler has achieved speedups from 2.8 to 98. Since the PIII's clock frequency is 6.2 times faster than the considered XPP, the results are very promising. Note that no benchmark was specially rewritten to exploit the XPP architecture more efficiently (*e.g.*, partitioning and distribution of arrays among the internal memories) and thus some results could be further improved. Also note that there are other advantages to use the XPP instead of a microprocessor (*e.g.*, power consumption reduction).

All benchmarks were compiled in less than 6 seconds, including placement and routing. This shows that it is possible, when targeting the XPP platform, to have compilation runtimes comparable to the ones achieved by software compilation, and to still accomplish efficient implementations.

<sup>7</sup> The internal XPP RAMs keep, by default, their data during reconfiguration.

**Table 1.** Properties of the benchmarks used.

Benchmark	Source	#loC	#loops	#arrays	primary data size	description
dct	in-house	90	8	4	512×512	8×8 block 2-D DCT (based on matrix multiplications) traversing an image
fir	in-house	18	2	3	10,000	1D FIR filter with 32 taps
bpic	COSYMA	151	8	5	512×512	binary pattern image coding algorithm
edge_det	UTDSP	188	16	6	512×512	edge detector with smooth (3×3) and using vertical and horizontal 2-D convolutions
life	Rawbench	118	10	2	8×8	Conway's Game of Life (4 iterations)
fdct	TI C62x <sup>1m</sup> DSP	228	3	3	512×512	8×8 block 2-D FDCT with rounding (pointer-free version)
median	MIT Bitwise	96	4	6	10,000	median filter (5×9)

**Table 2.** Results obtained with XPP-VC.

Benchmark	loop unrolling	Compilation time (sec)	#Part	#PEs	#Max	computation (#ccs)	total (#ccs)	Speedup
dct	yes	2.2	1	240	26	11,239,424	11,252,333	2.9
fir	yes	0.9	1	132	67	11,942	16,853	40.0
bpic	yes	3.4	1	237	12	945,984	1,047,733	2.8
edge_det	yes	5.6	5	54/225	12	3,536,384	3,541,818	9.3
fdct	no	1.8	2	150/299	14	1,231,872	1,240,078	12.7
median	yes	1.3	2	201/231	44	22,906	28,090	98.0

**Table 3.** Results using *loop dissevering*.

Benchmark	w/o loop dissevering			w/ loop dissevering			Ratio	
	#Part	#PEs	total (#ccs)	#Part	#PEs	total (#ccs)	PEs	ccs
dct (16 8×8 blocks)	1	123	148,215	5	54/132	152,182	0.44	1.02
bpic (16×16 image)	1	148	13,752	5	97/189	32,737	0.66	2.38
life	4	185/317	124,286	8	102/325	113,698	0.55	0.92

In order to show the capabilities of the *loop dissevering* transformation, Table 3 presents the results obtained with `dct`, `bpic`, and `life`. The technique was applied in order to allow mapping to a smaller array. Forcing *loop dissevering*, the implementations needed from 34% to 56% fewer PEs. `dct` and `life` have not exhibited significant differences in the execution time after applying *loop dissevering*. However, `bpic` needed over twice the number of clock cycles. This performance reduction is related to the configuration overhead, when the computation time on configurations called several times is not long enough. Such configuration overhead could almost be neglected on architectures with several on-chip configuration contexts. In this case we predict that *loop dissevering* will only result in minor performance reductions.

## 6 Related Work

We now compare briefly our high-level compilation and temporal partitioning approach with closely related work.

**Compilation:** High-level compilation for reconfigurable logic has been the focus of many researchers since the first attempts [7]. Most of this work targets fine-grain devices (e.g. FPGAs) which require logic synthesis and/or module generators. In addition, fine-grain architectures also need very time consuming backend phases (mapping and place and route) and iterative refinements over the entire compilation flow. Even when pre-placed and pre-routed components are used, the compilation time is still in the order of minutes or hours. Since XPP is a coarse-grained architecture, which directly supports arithmetic and other operations occurring in high-level languages, complex synthesis and mapping phases are not needed.

XPP-VC uses the *pipeline vectorization* method presented in [6], while other compilers such as Garp-C [9] and Napa-C [8] are based on software pipelining techniques. The control structure generation, based on the event handling mechanisms of the XPP, is completely new. Those structures are also directly mapped to XPP resources handling events.

**Temporal Partitioning:** Temporal partitioning at the behavioral level has already been successfully conducted on FPGAs [11]. The temporal partitioning algorithm used in the XPP-VC compiler is based on some ideas presented in [10]. *Loop dissevering*, a new method for partitioning loops, is also proposed in this paper. Previous approaches, which target FPGAs, consider a type of loop distribution for loops not fitting in the available resources [11]. However, large loops which cannot be distributed cannot be compiled. Our method can handle programs with unlimited complexity, provided only the supported C subset is used, and is always applicable.

## 7 Conclusions and Future Work

We have described the new Vectorizing C Compiler XPP-VC which maps programs in a C subset extended by port access functions to PACT's XPP architecture. Assisted with a fast place and route tool, it provides a complete “push-button” path from algorithms to XPP configuration binaries with short compilation times. Speedup factors of upto 98 for a 150MHz XPP core over a 933MHz PIII have been achieved, using this compiler. It integrates both temporal partitioning and a new loop partitioning mechanism, which allows to map large programs containing hundreds of lines of code. Since the XPP includes a configuration manager, programs are compiled to a single, self-contained XPP binary file that can be executed without assistance of any external host processor, even when several configurations are used. Our evaluation showed that XPP reconfiguration is fast enough to make runtime reconfiguration (RTR) worthwhile if medium-size data sets are processed between reconfigurations. Work on the XPP architecture to accelerate reconfiguration considerably, so that RTR is also worthwhile with small data sets processed between reconfigurations, is being carried out.

Ongoing work on the XPP-VC focuses on optimizing the compilation, on circumventing the current restrictions for pipelining loops, and on supporting other loop transformations. We are also researching a temporal partitioning algorithm which will consider the overlapping of fetching, configuring, and computation stages as well as configuration prefetching techniques to minimize the overall execution time. Furthermore, a future extension of the compiler for a host-XPP hybrid system is planned.

## References

1. R. Hartenstein, "A Decade of Reconfigurable Computing: a Visionary Retrospective," In *Proc. Design, Automation and Test in Europe*, 2001, pp. 642-649.
2. PACT Informationstechnologie GmbH, Germany, "The XPP White Paper," Release 2.0, June 2001. <http://www.pactcorp.com>
3. V. Baumgarte, et al., "PACT XPP - A Self-reconfigurable Data Processing Architecture," In *Journal of Supercomputing*, Kluwer Academic Publishers, (to appear).
4. SUIF Compiler system, <http://suif.stanford.edu>
5. Muchnick, S. S., *Advanced Compiler Design and Implementation*, Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1997.
6. M. Weinhardt and W. Luk, "Pipeline Vectorization," In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Feb. 2001, pp. 234-248.
7. Ian Page and Wayne Luk, "Compiling occam into FPGAs," In *FPGAs*, Will Moore and Wayne Luk, editors, Abingdon EEI&CS Books, 1991, pp. 271-283.
8. Maya Gokhale, and Edson Gomersall, "High-Level Compilation for Fine Grained FPGAs," In *Proc. IEEE 5th Symposium on Field-Programmable Custom Computing Machines (FCCM'97)*, Napa Valley, CA, USA, April 16-18, 1997, pp. 165-173.
9. T. J. Callahan, J. R. Hauser, and J. Wawrzynek, "The Garp architecture and C compiler," In *IEEE Computer*, 33(4), April 2000, pp. 62-69.
10. João M. P. Cardoso, "A Novel Algorithm Combining Temporal Partitioning and Sharing of Functional Units," In *Proc. IEEE 9th Symposium on Field-Programmable Custom Computing Machines (FCCM'01)*, Rohnert Park, CA, USA, April 30 - May 2, 2001.
11. Meenakshi Kaul, Ranga Vemuri, Sriram Govindarajan, Iyad E. Ouass, "An Automated Temporal Partitioning and Loop Fission approach for FPGA based reconfigurable synthesis of DSP applications," in *Proc. IEEE/ACM Design Automation Conference (DAC'99)*, New Orleans, LA, USA, June 21-25, 1999, pp. 616-622.