Int J Parallel Prog
DOI 10.1007/s10766-010-0132-7

ACOTES Project: Advanced Compiler Technologies for Embedded Streaming

Harm Munk · Eduard Ayguadé · Cédric Bastoul · Paul Carpenter · Zbigniew Chamski · Albert Cohen · Marco Cornero · Philippe Dumont · Marc Duranton · Mohammed Fellahi · Roger Ferrer · Razya Ladelsky · Menno Lindwer · Xavier Martorell · Cupertino Miranda · Dorit Nuzman · Andrea Ornstein · Antoniu Pop · Sebastian Pop · Louis-Noël Pouchet · Alex Ramírez · David Ródenas · Erven Rohou · Ira Rosen · Uzi Shvadron · Konrad Trifunović · Ayal Zaks

Received: 8 February 2009 / Accepted: 23 February 2010

Abstract Streaming applications are built of data-driven, computational components, consuming and producing unbounded data streams. Streaming oriented systems have become dominant in a wide range of domains, including embedded applications and DSPs. However, programming efficiently for streaming architectures is a challenging task, having to carefully partition the computation and map it to processes

H. Munk (✉) · Z. Chamski · P. Dumont · M. Duranton
NXP Semiconductors, Eindhoven, The Netherlands
e-mail: munkharm@xs4all.nl

R. Ladelsky · D. Nuzman · I. Rosen · U. Shvadron · A. Zaks
IBM Haifa Research Laboratories, Haifa, Israel

C. Bastoul · A. Cohen · P. Dumont · M. Fellahi · C. Miranda · L.-N. Pouchet · K. Trifunović
Alchemy Group, INRIA Saclay and LRI, Paris-Sud 11 University, Paris, France

M. Cornero · A. Ornstein · E. Rohou
STMicroelectronics, Cornaredo, MI, Italy

E. Ayguadé · P. Carpenter · R. Ferrer · X. Martorell · A. Ramírez · D. Ródenas
Universitat Politècnica de Catalunya, Barcelona, Spain

M. Lindwer
Silicon Hive, Eindhoven, The Netherlands

A. Pop
Centre de Recherche en Informatique, MINES ParisTech, Paris, France

S. Pop
Compiler Performance Engineering, Advanced Micro Devices, Austin, TX, USA

Published online: 20 April 2010

in a way that best matches the underlying streaming architecture, taking into account the distributed resources (memory, processing, real-time requirements) and communication overheads (processing and delay). These challenges have led to a number of suggested solutions, whose goal is to improve the programmer's productivity in developing applications that process massive streams of data on programmable, parallel embedded architectures. StreamIt is one such example. Another more recent approach is that developed by the ACOTES project (Advanced Compiler Technologies for Embedded Streaming). The ACOTES approach for streaming applications consists of compiler-assisted mapping of streaming tasks to highly parallel systems in order to maximize cost-effectiveness, both in terms of energy and in terms of design effort. The analysis and transformation techniques automate large parts of the partitioning and mapping process, based on the properties of the application domain, on the quantitative information about the target systems, and on programmer directives. This paper presents the outcomes of the ACOTES project, a 3-year collaborative work of industrial (NXP, ST, IBM, Silicon Hive, NOKIA) and academic (UPC, INRIA, MINES ParisTech) partners, and advocates the use of Advanced Compiler Technologies that we developed to support Embedded Streaming.

Keywords Parallel architectures · Compilers · Streaming applications · Automatic Parallelisation · HiPEAC

1 Introduction

Streaming applications which dominantly process large amounts of data have increasing demands for processing power. This demand stems from several requirements: on the one hand, the amount of processing per data element increases because of higher quality requirements of the result (e.g. video processing). On the other hand, the amount of data per unit of time also increases (e.g. higher communication speeds in wireless networks). This, in fact, calls for higher silicon efficiency, a demand that was met up to a few years ago by designing application specific integrated circuits (ASICs). The time to design such ASICs is, however, proportional to the complexity of the ASIC; as the complexity of the ASIC grows exponentially, their design becomes economically infeasible. Designers have thus shifted their focus toward programmable platforms, thereby potentially amortizing the design cost across several applications, or even application domains. Programmable platforms have traditionally been unable to meet the high throughput requirements: they were mostly designed for general purpose computation and offering limited parallelism opportunities.

Several recent architectures do expose parallelism to the application programmer. This, however, shifts the problem of managing complexity partly from the hardware designer to the software application developer. Exploiting available parallelism optimally requires intimate knowledge of both the application and the target platform. Automating the extraction of parallelism from sequential algorithmic descriptions has proven to be an extremely complex task in general.

In 2006, IBM, Philips (later: NXP Semiconductors), STMicroelectronics, NOKIA, INRIA and UPC initiated the ACOTES project to advance the support of application programmers in parallelising applications on highly parallel architectures. They were later joined by Silicon Hive and MINES ParisTech. The ACOTES project concentrates on developing tools to assist the application programmer in achieving optimal parallelism. From the outset we decided to use a mainstream language (C), an existing compiler framework (the GNU Compiler Collection—GCC), focus on the data streaming application domain, and target three distinct state-of-the-art multicore architectures (Cell Broadband Engine, xSTream processor, and Ne-XVP). This way we were able to concentrate our efforts on support for parallelization across several levels. Data streaming applications typically contain potential for both coarse-grain task-level parallelism across threads, fine-grain data-level parallelism residing inside nested loops of SIMD-like computations, and also memory-level parallelism to optimize data transfers.

1.1 Applications

The ACOTES project focuses on the data-streaming application domain. In this paper we present experimental results using three applications from this domain: FM-radio, H264 and Gamma-correction. The project, however, uses several additional streaming applications to drive the developments in the project.

1.1.1 FMradio

The FM-radio application was extracted and adapted from the GNU Radio project [1]. It contains about 500 lines of code. The application receives an input stream, applies a number of filters to it, and finally writes an output stream. Several of the filters apply the same transformation with different configuration parameters. The structure of the filters is shown in Fig. 1. The *FFD* filter is the most time consuming one.

1.1.2 H.264

The H.264 application is part of the MPEG-4 standard [2]. It consists of a video and audio coder and decoder, achieving high levels of compression for improved transmission and storage of streaming media files. We study a subset of its internal algorithms to demonstrate the vectorization capabilities introduced in GCC.

1.1.3 Gamma Correction

The Gamma correction algorithm is one of the last phases in a typical image processing pipeline. It features a triply nested loop scanning over the pixels of a 2D image. For each pixel, it searches through an array of thresholds until it finds the threshold interval (plotted along the X-axis of Fig. 2) within which the color value x lies. The new pixel value is inferred from the offset, gradient and threshold of the interval. The signal processing flow is depicted in Fig. 3.

The current customer requirement from the Gamma correction algorithm calls for pixel throughtput of about four cycles-per-pixel for the whole image signal processing

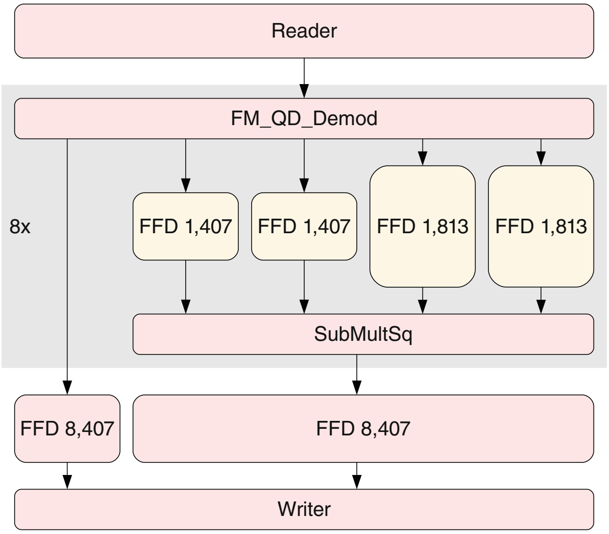


Fig. 1 FMradio filters structure

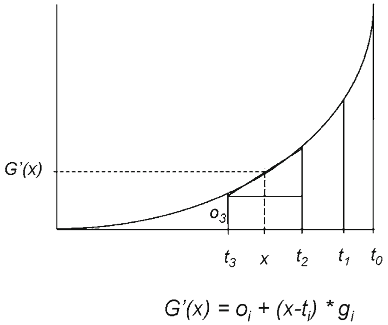


Fig. 2 Gamma correction filter

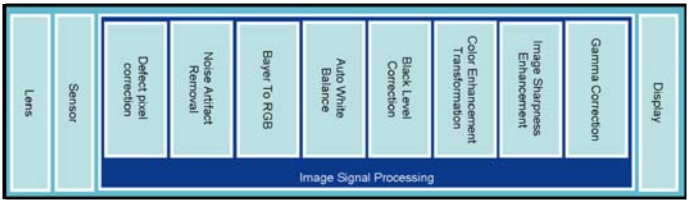


Fig. 3 Image signal processing algorithm pipeline

pipeline. A straightforward scalar implementation of Gamma correction takes 154 cycles per pixel. This application illustrates the potential of outer-loop vectorization, developed within ACOTES.

1.2 Compiler Structure

GCC, the GNU Compiler Collection is a compiler framework supporting several mainstream programming languages as well as a large array of target processors. The framework is actively maintained by the GCC project of the Free Software Foundation, supervised by the GCC steering committee. The project consists of a mainstream development part, which results in regular releases, and several development parts often dubbed development branches. The mainstream development is closely supervised by the GCC steering committee delegating responsibilities to leading developers. Results from development branches are incorporated into the mainstream only after scrutiny and approval by designated maintainers. This setup allows for the simultaneous development of a production quality compiler and a set of new, experimental prototypes.

The structure of GCC itself is that of a traditional compiler: a front-end for language parsing and semantics checking, a "middle-end" for target-independent optimisations, and a back-end for code generation and target-specific optimisations. It is this combination of support for mainstream languages and targets, industrial mainstream quality, the ability to experiment freely in parallel to the main line of development without interference, a vibrant supporting community and its adoption by a vast number of users that makes GCC an ideal choice for a project such as ACOTES.

1.3 ACOTES Project Structure

The ACOTES project is divided into a set of subtopics:

- **Abstract Streaming Machine.** In order to target several distinct architectures, we developed an abstract machine model that captures the streaming characteristics of the platforms in a common model called the Abstract Streaming Machine, or ASM.
- **Streaming Programming Model.** Starting from the C-language, we developed a number of extensions that express parallelism opportunities in the source, collectively referred to as the Streaming Programming Model, or SPM. As an extra requirement, these language extensions had to be neutral with respect to the core C language: the pragma mechanism is ideal for this purpose. This decision, however, surfaced a shortcoming of GCC, which is not very well suited for experimenting with front-end transformations. This prompted the project to turn to Mercurium [3] as a source-to-source prototyping tool for implementing the SPM.
- **Middle-end Optimisations.** Streaming applications are characterized by nested loops of computations. Considerable performance improvements can be gained by applying optimisations to these loops. The project concentrated its effort on the polyhedral model for high-level loop optimisations in the middle-end.
- **Back-end Optimisations.** Modern architectures often contain SIMD-like instructions, also called vector instructions. As these instructions are target specific, opportunities for applying them are detected in the back-end of the compiler.

- **Code Generation.** Code generation is typically specific to each architecture; in the ACOTES project, we therefore decided to concentrate our effort on common back-end algorithms and on common back-end formats.

Apart from the technical topics, the project has a separate activity to disseminate its results, part of which is maintaining contact with the GCC and HiPEAC network of excellence communities.

This article describes the design and implementation of the ACOTES tool-chain. Section 2 describes related work. Sections 3 and 4, respectively, present the Abstract Streaming Machine (compilation target) and the Streaming Programming Model (extensions to the ISO-C language). Section 5 presents the automatic loop nest optimisations and vectorisation, and the interaction between them. Section 6 describes the target platforms and the code generation phase of the compiler for these architectures. Section 7 presents some experimental results, and Sect. 8 concludes.

2 Related Work

The ACOTES project takes a holistic approach to parallel stream-programming, spanning over the whole flow of compilation down to the runtime system. It has connections with a large number of related work. This section compares our approach with the most closely related results in the field of high-performance embedded computing.

StreamIt is a long running project with a publicly available compiler and benchmark suite. The StreamIt [4] language imposes a hierarchical structure on the program composed of filters, pipelines, split-join operators and feedback loops. It requires the developer to structure the program into separate work functions per filter, in contrast to using pragmas which maintain the original structure of the code. The StreamIt compiler [5] targets the Raw Architecture Workstation, symmetric and heterogeneous multicore architectures, and clusters of workstations, where aggressive task-level optimizations are performed automatically [6]. StreamIt does not employ a generic machine model like the ACOTES ASM, and the ACOTES SPM is much more expressive than the cyclostatic data-flow model of computation underlying StreamIt [7], while still facilitating compilation-time task-level optimizations [8,9]. Compared to StreamIt, our approach involves a tight coupling of task- and loop-level optimizations, enabling more relevant decisions about synchronization adaptation, communication, multi-level exploitation of parallelism and locality. The optimization strategy relies on iterative search which helps us find interesting tradeoffs inaccessible to partitioned compilation flows separating these problems into different representations and passes [10]. We will illustrate this design on the interplay between task-level and loop-level optimizations, including automatic vectorization.

Sequoia is a well known data parallel language exploiting the structure of data-centric algorithms [11]. It severely restricts the model of computation to hierarchical fork-join parallel sections, but allows the programmer to state data affinity to portions of iteration space. For each hardware platform, the application programmer must supply a mapping that takes the “abstract” hierarchy defined in the application, and assigns pieces of it onto specific hardware. This approach requires more effort from the application providers and requires them to learn the memory characteristics of each hardware platform, but it is certainly a pragmatic solution that could be added as an optional feature to the ACOTES programming model.

The new specification of OpenMP [12,13], version 3.0, supports task parallelism using the new `task` directive. This directive specifies that the serial code within it can be executed by another thread inside the scope of a `parallel` region. In OpenMP, every time the task directive is reached a new task is created to execute its body. In the ACOTES SPM, all the inner tasks are created once when the `taskgroup` directive is reached, and a value is sent on each input stream each time the task directive is reached. This is a form of synchronization that does not exist in the OpenMP 3.0 proposal. However, there are other proposals for OpenMP that add synchronization between threads. Gonzalez et al. [14,15] propose three new directives: PRED, SUCC and NAME. The NAME directive labels a worksharing, and this label can be used by PRED and SUCC directives to specify synchronization. Another approach using annotated C is Cell Superscalar (CellSs) [16], which uses a `task` directive to express what are the inputs and outputs at a function level. Each time the function is called, a new task is created and the runtime system takes care of the possible dependencies it may have with other tasks.

StarPU features a stream-oriented model of computation, but focuses on the dynamic sheduling aspects and does not involve any language extension [17]. Based on multi-versioned kernels, it automates the dynamic balancing and mapping of tasks and data over heterogeneous, accelerator-centric parallel architectures. It would be an interesting target for GCC and the ACOTES runtime system.

StreamRoller [6] is a stream compiler for the Cell Broadband Engine, which uses the SGMS algorithm to split stateless kernels, partition the graph, and schedule it statically. Task fission and fusion are translated into an Integer Linear Programming (ILP) problem, which is solved using the commercial CPLEX solver [18].

Gedae is a proprietary graphical programming environment for streaming signal processing applications in the defense industry. Unlike ACOTES, the developer specifies the mapping of the program onto the target, and the compiler generates the executable according to this mapping [19].

We selected GNU Radio as our motivating example [1]. It is a framework developed in C++ and Python. GNU Radio allows to express graphs of filters and connections described using Python. Filters are usually constructed as C++ classes. GNU Radio comes with its own task scheduler and the system can be deployed on multiple architectures, including even FPGAs. GNU Radio provides more than 100 different basic blocks that can be combined to achieve the goal of the application. New blocks may be added to add new functionality. Both StreamIt and GNU Radio are designed for signal processing applications, and require the program to be written specifically in terms of streaming blocks.

3 ASM

The Abstract Streaming Machine [20,21] is the compiler's description of the target multiprocessor system. The ASM defines the search space for the compiler's partitioning and allocation algorithm by specifying the system topology and performance

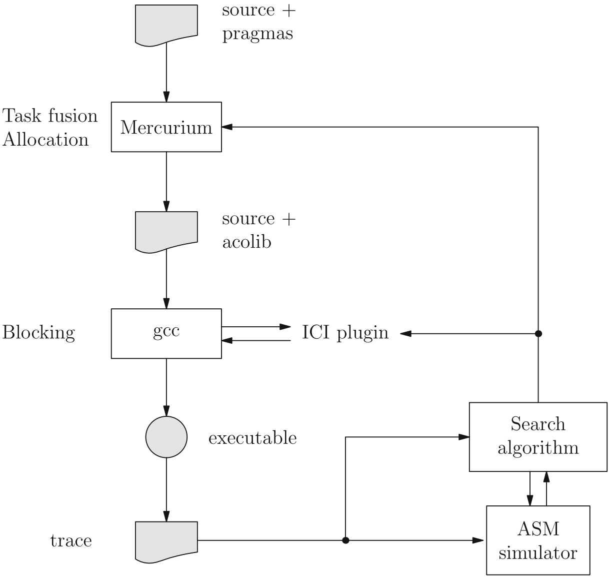


Fig. 4 The feedback loop of the ACOTES compiler: a heuristic search algorithm controls Mercurium and GCC

characteristics, and providing constraints on the mapping such as allowed memory sizes.

We have implemented the ASM in a coarse-grain simulator, which estimates the performance of a candidate mapping on the given target without iterative compilation and execution on a real platform. In addition to improving the speed of the compiler, the simulator allowed us to initiate work on the search algorithm before the compiler's transformation infrastructure is complete, and is repeatable because there are no experimental errors.

Figure 4 shows the structure of the ACOTES compiler, including the ASM simulator. The compilation flow is iterative: a heuristic search determines a candidate mapping which is compiled using Mercurium [3] and GCC. The mapping is provided to GCC through a plugin using the interactive compilation interface (ICI) [22].

3.1 The ASM Machine Model

The topology of the target platform is given by an unstructured bipartite graph $H = (V, E)$ where $V = P \cup M \cup I$ is the set of vertices, a disjoint union of processors, P, and memories, M, in one partition and interconnects, I, in the other. The edges, E, serve only to define the topology. Figure 5 shows two example targets: (a) a Cell Broadband Engine, and (b) a four-core shared memory machine. Each processor, interconnect, and memory is defined using the parameters summarized in Figs. 6, 7 and 8, described in detail below. Figure 6 and 7 give the machine descriptions for the example targets, measured on Cell and estimated for a 4-processor SMP. The ASM defines the

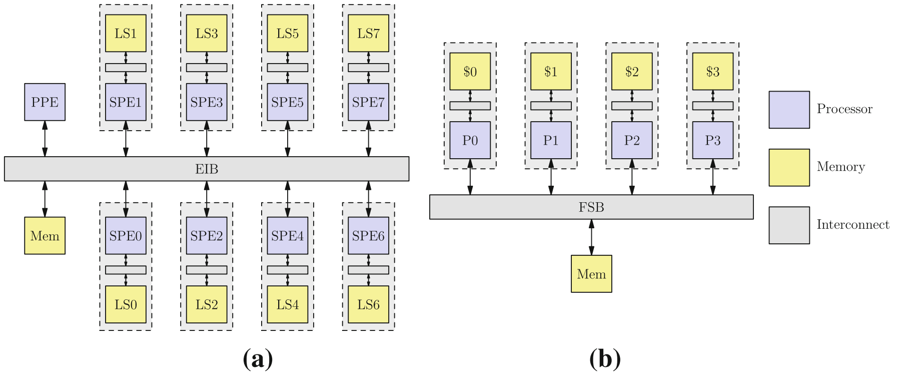


Fig. 5 Topology of two example targets for the ASM. **a** Cell-based system. **b** Four-core symmetric multiprocessor (SMP)

Parameter	**Type**	**Description**	**Cell**	**SMP**
`name`	String	Unique name in platform namespace	'SPE*n*'	'CPU*n*'
`clockRate`	Fixed pt.	Clock rate, in GHz	3.2	2.4
`hasIO`	Bool	True if the processor can perform IO	False	True
`addressSpace`	List	List of the physical memories in the system that are addressable by this processor and their virtual address	[('LS*n*',0)]	[('Mem',0)]
`pushAcqCost`	Int	Cost, in cycles, of acquiring a producer buffer (before waiting)	448	20
`pushSendFixedCost`	Int	Fixed cost, in cycles, of pushing a block (before waiting)	1104	50
`pushSendUnit`	Int	Number of bytes per push transfer unit	16384	0
`pushSendUnitCost`	Int	Incremental cost, in cycles, to push pushUnit bytes	352	0
`popAcqFixedCost`	Int	Fixed cost, in cycles, of popping a block (before waiting)	317	50
`popAcqUnit`	Int	Number of bytes per pop transfer unit	16384	0
`popAcqUnitCost`	Int	Incremental cost, in cycles, to pop `popUnit` bytes	0	0
`popDiscCost`	Int	Cost, in cycles, of discarding a consumer buffer (before waiting)	189	20

Fig. 6 Definition of a processor

machine characteristics that are visible to software, including the ACOlib runtime, so it may not exactly match the underlying physical hardware. For example, the Operating System in a Playstation 3™makes only six SPEs available to software, and the mapping from virtual to physical core is not known. We assume that any processors available to the stream program will not be time-shared with other applications while the stream program is running.

Parameter	Type	Description	Cell	SMP
`name`	String	Unique name in platform namespace	`'LS`*n*`'`	`'Mem'`
`size`	Int	Size, in bytes	262144	2147483648
`clockRate`	Fixed pt.	Clock rate, in GHz	`3.2`	`0.4`
`latency`	Int	Access latency, in cycles	2	4
`bandwidth`	Int	Bandwidth, in bytes/cycle	128	8

Fig. 7 Definition of a memory

Parameter	Type	Description	Cell	SMP
`name`	String	Unique name in platform namespace	`'EIB'`	`'FSB'`
`clockRate`	Fixed pt.	Clock rate, in GHz	`1.6`	`0.4`
`elements`	[String]	List of names of the elements (processors and memories) on the bus	`['PPE', 'SPE0', ···, 'SPE7']`	`['CPU0', ···, 'CPU3']`
`interfaceDuplex`	[Bool]	If the bus has more than one channel, then define for each processor whether it can transmit and receive simultaneously on different channels	`[True, ···, True]`	`[False, ···, False]`
`interfaceRouting`	[Enum]	Define the routing type for each processor: *storeAndForward*, *cutThrough*, or *None*	`[None, ···, None]`	`[None, ···, None]`
`startLatency`	Int	Start latency, L cycles	80	0
`startCost`	Int	Start cost of the channel, S cycles	0	0
`bandwidthPerCh`	Int	Bandwidth per channel, B bytes/cycle	16	16
`finishCost`	Int	Finish cost, F cycles	0	0
`numChannels`	Int	Number of channels on the bus	3	1
`multiplexable`	Int	False for a hardware FIFO that supports only one stream	`True`	`True`

Fig. 8 Definition of an interconnect

Each processor is defined using the parameters in Fig. 6. The ASM supplements the back-end compiler machine description: the details of the processor ISA and micro-architecture are not duplicated in the ASM. The `addressSpace` and `hasIO` parameters provide constraints on valid mappings. The former defines the local address space of the processor, i.e., which memories are directly accessible through ordinary load-store instructions, and where they appear in virtual memory; it is used to place stream buffers. The latter defines which processors can perform system IO, and is a simple way to ensure that tasks needing system IO are mapped to appropriate processors.

Each memory is defined using the parameters in Fig. 7. The latency and bandwidth figures may be used by the compiler to refine the estimate of the execution time of each task. The memory sizes are used to determine where to place communications buffers, and provide constraints on loop blocking factors.

Each interconnect is defined using the parameters shown in Fig. 8. The graph topology is given by the `elements` parameter, which lists the processors and memories that are adjacent to the bus. Each interconnect is modelled as a bus with multiple channels, which has been shown to be a good approximation of the performance observed in practice when the processors and memories on a single link are equidistant [23]. Each bus has a single unbounded queue to hold the messages ready to be transmitted, and one or more channels on which to transmit them. Streams are statically allocated onto buses, but the choice of channel is made at runtime. The `interfaceDuplex` parameter defines for each processor or memory whether it can simultaneously read and write on different channels. The bandwidth and latency of each channel is controlled using four parameters: the start latency (L), start cost (S), bandwidth (B) and finish cost (F). The latency of transferring a message of size n bytes is given by $L + S + \lceil \frac{n}{B} \rceil$ and the cost incurred on the link is $S + \lceil \frac{n}{B} \rceil + F$. This is a natural model for distributed memory machines, and is equivalent to the assumption of cache-to-cache transfers on shared memory machines.

The ASM simulator assumes that the only significant traffic on an interconnect is the transfer of messages related to streams. Hence each processor should have some private memory—either a local store or a cache. If it is a local store, the compiler must allocate the stream buffers in this local store [24]. If it is a cache, the ASM assumes that it is sufficiently effective so that the cache miss traffic on the interconnect is low.

Hardware routing is controlled using the `interfaceRouting` parameter, which defines for each processor whether it can route messages from this interconnect onto another interconnect that it is adjacent to. Each entry can take the value `storeAndForward` (receive a complete message and check its integrity before forwarding it down the route), `cutThrough` (start forwarding a message before it is complete, increases throughput at the expenses of reliability), or `None` (no routing capability).

3.2 The ASM Program Model in the Simulator

The coarse-grain simulator models the stream program as a directed graph $G = (T, S)$ where T is the set of vertices representing tasks and S is the set of edges representing streams. The graph does not have to be acyclic, but it must be connected (simulation of a single streaming application).

Note that a task may have irregular data-dependent behaviour. We therefore divide tasks into *subtasks*, which are the basic unit of sequencing. A subtask pops a fixed number of elements from each input stream and pushes a fixed number of elements into each output stream. In detail, the work function for a subtask is divided into three consecutive phases: first, the *acquire phase* calls `Iport_acquire` or `Oport_acquire` for each stream to obtain the next set of full input buffers and empty output buffers. Second, the *processing phase* works locally on these buffers, and is modelled using a fixed or normally-distributed processing time. Finally, the *release phase* calls `Iport_pop` to discard the input buffers, and calls `Oport_push` to send the output buffers, releasing the buffers in the same order they were acquired. This three-stage model is not a fundamental requirement of the ASM, and was introduced as a convenience in the implementation of the simulator; the ACOTES compiler generates subtasks of this form.

Streams are defined by the size of each element, the location and the length of the distinct producer and consumer buffers (distributed memory) or the single shared buffer (shared memory). These buffers do not have to be the same length. Streams are point-to-point, so each stream has exactly one producer task and one consumer task, but those tasks may access the same stream from more than one subtask (precise semantics and examples will be presented in the next section).

3.3 Definition and Sequencing of Irregular Tasks

The coarse-grain simulator uses the sequential semantics of the SPM program to control the sequencing of subtasks in the stream program. A task is controlled by its *subtask tree*, which is built up from subtasks, *If* nodes and *While* nodes. Each *If* or *While* node is associated with a similar statement in the sequential program.

When the simulator executes in the trace-driven mode, the executable is instrumented to record the outcome each time a control statement is executed. A control statement is an `if` or `while` statement in the original SPM program that controls one or more subtask trees. The resulting sequence of outcomes is known as a control variable, and takes the values 1 or 0 for an `if` statement, or the non-negative iteration count for a `while` statement. When the simulator is used in the trace-driven mode, the program model is driven by the set of control variables taken from the trace.

The set of control variables may be reused with a different partition or allocation. It usually cannot be reused with a different blocking factor, or after compiler transformations such as loop interchange or distribution, because of these transformation's impact on branch outcome statistics.

4 SPM and the Front-End Compiler

The streaming programming model (SPM) designed in the context of the ACOTES project is implemented using extensions to the C language. It consists of a set of *pragmas* extending the serial code semantics. The main requirements of the SPM are: to be easy to learn, easy to use and reuse, and to support task and data parallelism. We think that OpenMP [12] can therefore serve as a good basis to develop our SPM: OpenMP can be learned, applied, and tested incrementally, which is convenient for new programmers in the streaming field.

4.1 Elements in the SPM

The SPM adds three basic elements to OpenMP: *streaming tasks*, *streams*, and *ports*. These elements are enclosed into a *taskgroup* compound abstraction. Applications are represented in SPM as multiple tasks connected via point-to-point data streams. Each task may be viewed as an independent process with all its data being private. Communication and synchronization among tasks happen only via streams. A stream is directed, and we refer to its two end points (*ports* from now on) from the point of view of the task, so that the producer has an *output port* to generate data into the stream, and the consumer has an *input port* to read data from the stream. The two ends are permanently connected together. The consumer task blocks when it tries to read from an empty input stream, and the producer blocks when it tries to write to a full output stream.

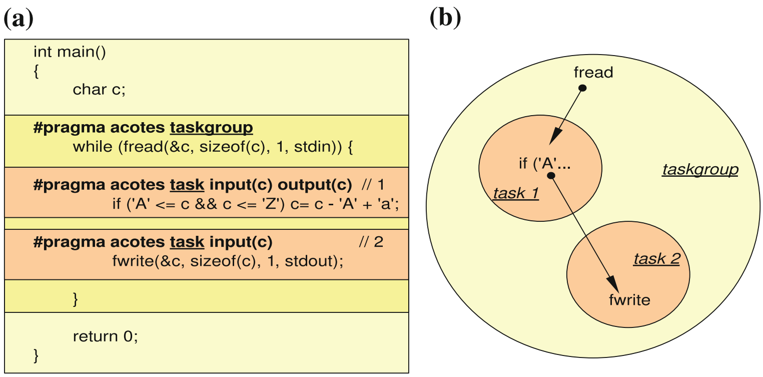
Using the scheme described above, task parallelism is supported in a traditional way with the addition of having communication channels between tasks. Data parallelism is supported through the ability of replicating a task into a number of instances, allowing to run each instance in a different core on disjoint input and/or output data.

There are several distinctions between the execution model of the SPM and that of OpenMP, namely:

- In the SPM, streaming tasks are created all at once when a *taskgroup* is entered. This is contrary to OpenMP where a thread creates each task in a parallel region upon encountering it dynamically.
- Tasks in SPM are permanent, meaning that they are alive while there is input data for them to process. This implies the automatic generation of an implicit loop: *while (there-is-input-data)* {. . .} , enclosing the code in the body of the task, contrary to what is done in OpenMP. Only when the input streams are known to have no more data, can the task finish. A *taskgroup* ends when all its tasks have finished.
- Contrary to OpenMP which supports shared data, data accessed by a task must be either private or acquired through an input stream of the task. The SPM defines specific situations where global data can be accessed through a well-defined interface.

4.2 Streaming Execution Model

An SPM program start executing serially as a single process. Upon entering a *taskgroup*, tasks are created and the program starts processing data in streams. Figure 9a shows a simple example, which converts an input stream read from *stdin* to lower case, and then writes the resulting stream to *stdout*. Figure 9b shows the scheme of tasks built using the SPM pragmas. As can be observed in the drawing, the *taskgroup* is used as a container for the annotated tasks. Arrows represent streams along the direction in which data circulates.



```
int main()
{
        char c;

#pragma acotes taskgroup
        while (fread(&c, sizeof(c), 1, stdin)) {

#pragma acotes task input(c) output(c)  // 1
                if ('A' <= c && c <= 'Z') c= c - 'A' + 'a';

#pragma acotes task input(c)            // 2
                fwrite(&c, sizeof(c), 1, stdout);

        }

        return 0;
}
```

Fig. 9 Example program (**a**), and representation of tasks (**b**)

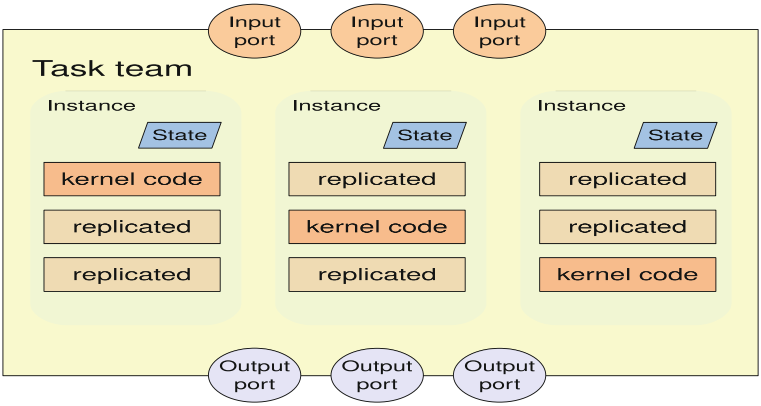


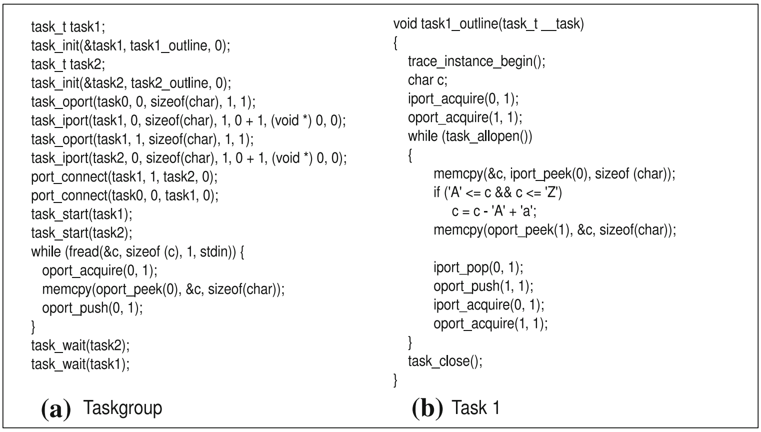
Fig. 10 Representation of a team of tasks

Observe that extra clauses are attached to the SPM task pragmas to express data transfers through the streams. The clauses *input* and *output* receive one variable for each input and output stream that should be established, respectively. A task can receive input data either from its taskgroup or from a previous task having the same variable designated as output. A task can send output data either to a subsequent task having the same variable designated as input, or to its taskgroup. This way, the taskgroup itself serves as an environment, used when there is no previous or subsequent task having the corresponding variable as output or input.

The SPM also supports the exploitation of data parallelism. In the streaming context, this implies processing chunks of data taken from streams in parallel. This is accomplished by replicating an SPM task into multiple instances (collectively called a *team*), where each instance processes a distinct set of input data chunks. Figure 10 depicts the execution of a team consisting of three instances. Tasks can have many inputs and many outputs. When splitting a task into a team of instances, each input stream can be split so that data chunks are distributed among the instances, or it can be replicated so that all instances will receive all data chunks. The team contribute to the output streams as a single task, where each output chunk is generated by a single instance at every task iteration. The runtime system implements this intertwined contribution by keeping track of which instance is the leader at every iteration. See in Fig. 10 how the complete kernel code (updating state and contributing output) is executed by a single instance at a time, while the code needed only for updating the state of each instance is executed in a replicated way.

4.3 Front-end Compiler and Runtime Library (ACOlib)

The transformations outlined above are implemented in the *Mercurium C Compiler* [3,21], which is a source-to-source compiler, generating C code. We use Mercurium to translate the full set of SPM pragmas and clauses into calls to a runtime library



```
task_t task1;
task_init(&task1, task1_outline, 0);
task_t task2;
task_init(&task2, task2_outline, 0);
task_oport(task0, 0, sizeof(char), 1, 1);
task_iport(task1, 0, sizeof(char), 1, 0 + 1, (void *) 0, 0);
task_oport(task1, 1, sizeof(char), 1, 1);
task_iport(task2, 0, sizeof(char), 1, 0 + 1, (void *) 0, 0);
port_connect(task1, 1, task2, 0);
port_connect(task0, 0, task1, 0);
task_start(task1);
task_start(task2);
while (fread(&c, sizeof (c), 1, stdin)) {
  oport_acquire(0, 1);
  memcpy(oport_peek(0), &c, sizeof(char));
  oport_push(0, 1);
}
task_wait(task2);
task_wait(task1);
```

(a) Taskgroup

```
void task1_outline(task_t __task)
{
  trace_instance_begin();
  char c;
  iport_acquire(0, 1);
  oport_acquire(1, 1);
  while (task_allopen())
  {
      memcpy(&c, iport_peek(0), sizeof (char));
      if ('A' <= c && c <= 'Z')
          c = c - 'A' + 'a';
      memcpy(oport_peek(1), &c, sizeof(char));

      iport_pop(0, 1);
      oport_push(1, 1);
      iport_acquire(0, 1);
      oport_acquire(1, 1);
  }
  task_close();
}
```

(b) Task 1

Fig. 11 Code generated from the taskgroup example

which supports task and stream management. The resulting C output is then compiled with GCC, and linked with the runtime library to obtain the final binary file.

The initial approach for a runtime system in the ACOTES project was to implement a simple library (ACOlib) supporting the functionality needed for tasking and streams. We developed an implementation of ACOlib for generic SMP environments, but designed it in such a way that the library could also work with possibly distributed memory architectures and local stores.

ACOlib supports streaming execution in SMP environments, as shown in Fig. 11a which depicts a snippet of the code generated for the taskgroup example presented in Fig. 9a. Observe how two tasks are initialized, their ports created and connected, and then they are started. The two tasks are alive while the taskgroup reads characters from the file and sends them to *Task 1*. Figure 11b shows the code resulting from the transformation of *Task 1* of the same example. This task reads characters from its input port, processes them, and writes the result onto its output port. The task will remain alive as long as there are characters available in its input stream.

4.4 Integration in OpenMP and GCC

Inspired by the full set of SPM pragmas and their support by Mercurium and ACOlib, we also developed a reduced version especially designed as a minimal extension [9] of the OpenMP3.0 standard [12]. This approach leverages the knowledge of OpenMP thereby improving the learning curve while preserving, to a certain extent, the semantic of the SPM pragmas. The additional compiler support required by this extension is implemented directly in GCC and its runtime library libGOMP.

In order to provide stream programming support in OpenMP, the minimal necessary extension is to allow the use of lastprivate clauses on task constructs, without changing the existing semantic of the lastprivate clause. The semantic of firstprivate and lastprivate clauses is very close to SPM's input and output clauses. The firstprivate clause corresponds to data that is consumed by the task (flows in), while the

```
int main()
{
  char c;
#pragma omp parallel
  {
#pragma omp single
    {
      while (fread (&c, sizeof (c), 1, stdin))
        {
#pragma omp task firstprivate (c) lastprivate (c)
          if ('A' <= c && c <= 'Z')
            c = c  'A' + 'a';

#pragma omp task firstprivate (c)
          fwrite (&c, sizeof (c), 1, stdout);
        }
    }
  }
  return 0;
}
```

Fig. 12 Implementation of the example of Fig. 9a using OpenMP extended annotations

lastprivate clause corresponds to data that is produced by the task (flows out). The explicit knowledge about data flow between tasks helps the compiler's static analysis and facilitates generating stream communication between tasks.

To illustrate the OpenMP extension as well as the similarity of this approach with the SPM, we propose in Fig. 12 an implementation of the example code of Fig. 9a using OpenMP3.0 extended annotations. We implemented this support for streams in GCC's libGOMP runtime library [25], which also simplifies the toolchain by removing the dependence on the source to source *Mercurium C Compiler*.

4.5 MSF: Multicore Streaming Framework

The ACOlib runtime library developed to support tasks and streams was implemented for generic SMP environments, i.e. with shared memory. In order to apply our streaming framework to distributed memory architectures such as that of the Cell/B.E., we are making use of an underlying multicore streaming framework (MSF). On one hand, MSF provides efficient support for code management, as the code that the tasks execute is preloaded in advance of its execution. And on the other hand, it provides efficient support for data transfers between tasks, which may be running on the same or on different processors, as well as with shared or distributed memory. The abstraction layer implementing such data transfers resembles that of ACOlib, offering stream-based communication.

The use of MSF in ACOTES augments the general tool-chain of ACOTES which starts with Mercurium translating SPM pragmas into ACOlib calls. Instead, for the Cell/B.E., it makes direct use of MSF facilities for setting up tasks in remote processors and establishing stream communications among them. Furthermore, MSF also provides the runtime management and scheduling of task executions according to available data in the heterogenous and distributed environment of the Cell/B.E.

MSF provides a generic programming model for parallel platforms and an API for programming directly using its facilities. Although its first implementation is on the Cell/B.E., MSF can be implemented on any programmable platform. Tasks can be directed to various processors based on their processing characteristics. On the Cell/B.E the target processors are either PPEs or SPEs. The tasks are then compiled by a back-end compiler that deals with standard sequential code, and vectorization can be applied to achieve better performance on the target processor. Once available on a specific platform the same front-end compilation techniques can be performed independently of the underlined architecture. Using the information provided by the ASM, applications can be optimized for each platform by changing runtime parameters that are used by MSF.

4.6 Silicon Hive's ISO-C Language Extensions, Underpinning SPM

Silicon Hive, one of the ACOTES partners, had already developed and used an inhouse C-compiler (HiveCC) prior to the project. In the course of the project, Silicon Hive adapted HiveCC to the developing insights of the project, providing feedback to the other partners. By using the blocking communication ports of ACOTES, we realized that synchronization (i.e. the blocking behaviour) and data communication should take place at different granularities. This allows the compiler to expose parallelism between communication and computation. This resulted in ISO-C language extensions for synchronisation, as discussed below. We illustrate these synchronisation mechanisms in the context of the Gamma correction algorithm (introduced in Sect. 1.1). In this context, we also make use of other language extensions, such as: attributes, built-in types, function inlining, operator overloading, and register structures, as illustrated in Fig. 13 (for reference, the unoptimized version, which is used as an input to the ACOTES auto-vectorizer is shown in Fig. 19). HiveCC supports several pragmas by which the programmer can steer the compiler to reveal parallelism. HiveCC also provides built-in preprocessor definitions which enable the programmer to develop code independent of the underlying system architecture.

Taken together, these language extensions are needed to efficiently implement SPM on Silicon Hive processors.

Type attributes specify aspects of variables such as location, access routes, and synchronization with volatile operations. *Expression attributes* allow the user to specify the relative timing of associated statements, aliasing relations between variables, and the mapping of code on specific function units. `MEM` and `SYNC_WITH` in Fig. 13 are example for such attributes. These define input/output as mapped on a specific memory and with which side-effect statements their accesses need to be synchronized.

Built-in Types allow HiveCC to generate non-ISO-C operations on non-ISO-C types. These types result from processors being defined with datapaths of arbitrary width (i.e. not restricted to the widths of ISO-C datatypes). The additional types are available as signed or unsigned integers of processor-defined width and vectors with processor-defined numbers of elements and element widths. If the associated operators

```
HIVE_TERNARY_OPERATOR( ?,  OP_vec_mux
                        ,  tvec, tflags, tvec, tvec )
HIVE_BINARY_OPERATOR ( >>, OP_vec_asrrnd
                         , tvec, tvec, tvec )
HIVE_BINARY_CALL     ( >,  static inline, isp_vec_gt_c
                        ,  tflags, tvec, int )
#define LINESZV LINESZ/VECSZ
SYNC_WITH(0) tvec MEM(VMEM) inBuf[2*LINESZV];
SYNC_WITH(1) tvec MEM(VMEM) outBuf[2*LINESZV];
...
  // Initialize correction coefficients:
  int ta[SN]={...}, oa[SN]={...}, ga[SN]={...};
  // double buffering; pre-fetch & sync with buf 1:
  signal_inputFormatter() SYNC(0);
  for( line=0; line<LINES_PER_FRAME; line++ ) {
    int buf=line&1;  // determine buffer
    // double buffering; pre-fetch next line:
    signal_inputFormatter() SYNC(0);
    // double buffering; wait for current line:
    wait_inputFormatter() SYNC(0);
    for( c=0; c<LINESZV; c++ ) {
      tvec x = inBuf[buf*LINESZV+c];
      int t=thrh[0], o=offs[0], g=grad[0];
      for( i=0; i<SN-1; i++ ) {
        tflags flag = x>ta[i]; // overloaded '>'
        // overloaded vector '?' operator:
        o = flag ? o : OP_vec_clone(oa[i+1]);
        g = flag ? g : OP_vec_clone(ga[i+1]);
        t = flag ? t : OP_vec_clone(ta[i+1]);
#       pragma hivecc unroll
      }
      // overloaded vector '>>' operator:
      outBuf[buf*LINESZV+c] = o+(x-t)*g>>SCALE;
#     pragma hivecc unroll=6, pipelining=0
    }
    // Signal output buffer full, sync with output buf.
    signal_DMA() SYNC(1);
    // Wait until DMA finished previous line.
    if(line) wait_DMA() SYNC(1);
  }
```

Fig. 13 Gamma-correction kernel optimized with Silicon-Hive's SPM

have been overloaded (see below), these types may be used in arithmetic expressions. In the optimized Gamma correction algorithm in Fig. 13, pixel colour components are mapped onto the vector type `tvec`. In the case of ISP2300 this is a vector of 32 elements, each element being 16 bits wide.

Custom Operations can be called as intrinsic functions, which is needed when the overloading mechanism cannot distinguish operations based on signature alone. In fact, all operations defined on Silicon Hive processors may be called as intrinsic functions. Besides intrinsics for regular and custom operations, HiveCC also provides a number of *Pseudo Operations*. These operations are interpreted by the simulator, which provides cycle counts and more.

Operator Overloading allows the programmer to assign normal operator symbols to functions, including intrinsic functions for specific processor operators. HiveCC supports overloading of unary, binary, ternary, and assignment signatures. The optimized Gamma correction code provides three examples of overloading (see Fig. 13).

By applying overloading to inline functions, the programmer can develop machine-independent code without loss of performance. This is illustrated in the following code example:

```
inline static tvec fmux(tvec c,tvec t, tvec e) {
# if HAS_mux
    return(OP_mux(c,t,e));
# else
    return c?t:e;
}
HIVE_TERNARY_CALL(?,static,fmux,tvec,tvec,tvec,tvec)
```

Register Structures are an extension to the ISO-C `register` qualifier. By adding this qualifier to a newly defined type, the programmer indicates that all elements of the type are to be mapped onto registers.

In addition to the direct mapping of tasks to hardware synchronizations and communications, HiveCC revisits the SPM for the static interleaving of tasks. In a very wide VLIW context, multiple tasks may be statically scheduled and parallelized over multiple issue slots, converting task-level parallelism into instruction-level parallelism. Unlike the dynamic scheduling of a conventional task pipeline, load balancing and communication/computation overlapping needs to be performed statically by the compiler. A specific syntax is introduced to let the programmer expose such opportunities, and to fine-tune register pressure and resource usage. Inspired by these challenges, the ACOTES project also developed related techniques to optimize code size of nested software pipelined loops versus memory usage and register pressure [26].

In conclusion, the new synchronisation mechanism is needed to allow data to be communicated at pixel granularity, while synchronisation takes place at line granularity (for instance). This way, rather than keeping all volatile loads and stores together, the programmer is free to communicate data when it becomes available (streaming) and the compiler is free to schedule the resulting loads and stores within the constraints imposed by the synchronisation mechanism, exposing maximal parallelism.

Notice that other type attributes and built-in types are needed in this context, because they allow the compiler to reason about the many different non-ISO-C datatypes that embedded processors may support. Lastly, the overloading mechanism supports the programmer in reasoning about his code, which is particularly important when communication and computation need to be alternated.

5 Compiler Middle-End and Loop-Level Optimizations

Through the SPM, the programmer exposes much of the pipeline and data parallelism in stream computations. Recall that the goal of the ACOTES project is to minimize the burden of manually adapting a stream computing program to a new architecture. This adaptation is partly managed by the runtime system for the SPM, and partly by the compiler middle-end. Program transformations are indeed necessary to adjust the synchronization grain, and to tune the exploitation of the architecture's memory hierarchy w.r.t. the temporal and spatial locality of the streaming program. One typically splits such transformations into task-level and loop-level optimizations. Both address parallelism, locality and specialization, but generally at different levels (the levels of the memory hierarchy and the levels of parallelism in the target). In the following, we focus on loop-level optimizations, although our design is extensible to task-level optimizations such as static task pipelining, fusion and blocking [5]. Such an extension requires an adequate intermediate representation of the task-level data flow; this work is still in progress, based on Feautrier's proposal to extend the polyhedral representation to the full network of tasks [8]. We will conclude this section with the study of some tradeoffs between the exploitation of thread-level parallelism, fine-grain SIMD parallelism (e.g., vectors) and memory locality. Those tradeoffs are particularly important when optimizing a streaming application for multiple levels of parallelism, and considering power efficiency and compute density metrics.

5.1 Loop-Nest Optimizations

Loop nest optimizations are important compiler transformations to fully exploit the features of a given architecture. Current compiler techniques using syntax-based representations and hard-wired pattern matching techniques are not able to provide peak performance on complex architectures with multiple on-chip processing units.

In order to enable long and complex sequences of program transformations, we rely on a powerful algebraic program representation called the *polyhedral model*, where a sequence of transformations is represented as a single affine mapping function.

In scientific and engineering applications, most of the execution time is spent in nested loops. The *polyhedral model* views a dynamic instance (iteration) of each program statement as an integer point in a well defined subspace called *polyhedron*. Basing on this representation a dependence graph is built, which represents dependences among pairs of statement execution instances (iterations in loops).

The polyhedral model is classically applicable to *static control parts* (SCoP), that is loop nests in which the data access functions and loop bounds are affine combinations of enclosing loop iteration variables and *global parameters*.

5.1.1 Polyhedral Representation of Programs

Each dynamic instance of a statement S is denoted by a pair $(S, \mathbf{i})$, where $\mathbf{i}$ is iteration vector which contains values for the loop indices of the enclosing loops, from outermost to innermost. If loop bounds are affine expressions of outer loop indices and global parameters (usually, symbolic constants representing problem size) then the set of all iteration vectors associated with statement S can be represented by polytope $\mathcal{D}_S$ which is called the *iteration domain* of statement S. Let $\mathbf{g}$ be the vector of *global* *parameters* (a.k.a. structural parameters). Let D_S denote the matrix collecting affine loop bound constraints, the iteration domain set $\mathcal{D}_S$ is defined by

$$\mathcal{D}_S = \left\{\mathbf{i} \mid \mathrm{D}_S \times (\mathbf{i}|\mathbf{g}|1)^t \geq \mathbf{0}\right\}$$

5.1.2 Polyhedral Dependences

Dependences in a SCoP are represented as a Dependence Graph (DG). DG is directed multigraph $\mathrm{DG} = (V, E)$ where each vertex represent a statement and each edge $e^{S_i \to S_j} \in E$ from S_i to S_j represents a dependence polyhedron from dynamic instance of S_i to dynamic instance of S_j. The dependence polyhedron is a subset of the cartesian product of iteration domains $\mathcal{D}_{S_i}$ and $\mathcal{D}_{S_j}$. Dependence polyhedron for edge e is denoted as $\mathcal{P}_e$.

5.1.3 Access Functions

For each statement S, we define two sets $\mathcal{W}_S$ and $\mathcal{R}_S$ of $(\mathtt{M}, f)$ pairs, each pair representing a reference to variable M being written or read in statement S; f is the *access function* mapping iterations in $\mathcal{D}_S$ to memory locations in M. f is a function of loop iterators and global parameters. The access function f is defined by a matrix F such that

$$f(\mathbf{i}) = \mathrm{F} \times (\mathbf{i}|\mathbf{g}|1)^t\ .$$

Subscript function returns a vector whose dimensionality is equal to the dimensionality of an array M.

5.1.4 Scheduling Function

Iteration domains define exactly the set of executed dynamic instances for each statement. However, this algebraic structure does not describe the order in which each statement instance has to be executed with respect to other statement instances [10]. A convenient way to express the execution order for each statement instance is to give each instance an execution date. It is obviously impractical to define all dates explicitly because the number of instances may be either very large or unknown at compile time. An appropriate solution is to define, for each statement S, a *scheduling* function θ_S mapping instances of S to multidimensional timestamps (vectors). For tractability reasons, we restrict these functions to be affine, and we will use matrix operations on homogeneous coordinates (additional dimension equal to the constant 1) to represent affine functions. For the sake of transformation composition or search space exploration [27], θ_S is often broken into dedicated blocks: a matrix A^S operating on iteration vectors, a vector β^S for static (multidimensional) statement ordering, and a matrix Γ^S to parameterize the schedule and to model pipelining:

$$\Theta_S = \left[\begin{array}{ccc|ccc|c} 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_0^S \\ \mathrm{A}_{1,1}^S & \cdots & \mathrm{A}_{1,d^S}^S & \Gamma_{1,1}^S & \cdots & \Gamma_{1,d_\mathbf{g}}^S & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_1^S \\ \mathrm{A}_{2,1}^S & \cdots & \mathrm{A}_{2,d^S}^S & \Gamma_{2,1}^S & \cdots & \Gamma_{2,d_\mathbf{g}}^S & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathrm{A}_{d^S,1}^S & \cdots & \mathrm{A}_{d^S,d^S}^S & \Gamma_{d^S,1}^S & \cdots & \Gamma_{d^S,d_\mathbf{g}}^S & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 & \beta_{d^S}^S \end{array}\right]$$

$$\theta_S(\mathbf{i}_S) = \Theta_S \times \begin{pmatrix} \mathbf{i}_S \\ \mathbf{g} \\ 1 \end{pmatrix}$$

As an example we will consider the pseudocode in Fig. 18a. The loop kernel is composed of three statements: S_1, S_2 and S_3. The iteration domains for statements S_1 and S_2 are the following:

$$D^{S_1} = \left[\begin{array}{c|cc|c} 1 & 0 & 0 & 0 \\ -1 & 1 & 0 & -1 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le M-1 \end{array}$$

$$D^{S_2} = \left[\begin{array}{cc|cc|c} 1 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & -1 \end{array}\right] \begin{array}{l} 0 \le i \\ i \le M-1 \\ 0 \le j \\ j \le K-1 \end{array}$$

(domain of S_3 is the same as that of S_2). Domain of statement S_1 has single iterator dimension (corresponding to iteration variable `i`), and two parameter dimensions (corresponding to `M` and `K`. Domain of statement S_2 has two iterator dimensions (corresponding to iteration variables `i` and `j`). There are no array data access function for statement S_1 because it does not access any array. Data access functions for statement S_2 are the following:

$$\mathcal{W}^{S_2} = \{\}$$
$$\mathcal{R}^{S_2} = \left\{ \begin{array}{l} \left(\|x\|, \left[1\ 1 \middle| 0\ 0 \middle| 0\right]\right), \\ \left(\|c\|, \left[0\ 1 \middle| 0\ 0 \middle| 0\right]\right) \end{array} \right\} \begin{array}{l} x[i+j] \\ c[j] \end{array}$$

There are no write accesses in statement S_2, while there are two read data accesses: one from array `x` and other from array `c`. Original(corresponding to original input code) scheduling functions for statement S_1 and S_2 are given:

$$\mathrm{A}^{S_1} = \begin{bmatrix} 1 \end{bmatrix} \qquad \mathrm{A}^{S_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$\beta^{S_1} = \begin{bmatrix} 0 & 0 \end{bmatrix}^t \qquad \beta^{S_2} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^t$$

$$\Gamma^{S_1} = \begin{bmatrix} 0 & 0 \end{bmatrix} \qquad \Gamma^{S_2} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$$

$$\text{i.e. } \Theta^{S_1} = \left[\begin{array}{c|c} 0 & 0 \\ 1 & 0 \\ 0 & 0 \end{array}\right] \qquad \text{i.e. } \Theta^{S_2} = \left[\begin{array}{cc|c} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{array}\right]$$

note that Γ^{S_1} and Γ^{S_2} are all zeros, since the schedule does not depend on global parameters. A^{S_1} and A^{S_2} are identity matrices.

Program optimization in polyhedral model is usually done in three steps: (1) static analysis of input program resulting in algebraic representation of static control loop nests and construction of dependence graph, (2) transformations of polyhedral abstraction (based on linear algebra and integer linear programming machinery) without touching syntax representation of original program, (3) code generation (going back into syntax representation). Note that step (3) is done only once: all transformations (sequences) operate on polyhedral (algebraic) representation.

The following table lists the main loop transformations that we can freely compose and parameterize in our framework; see Allen and Kennedy [28] for reference on those transformations and Girbal et al. [27] for details on the encoding and composition invariants:

Transformation name	Matrices involved
Interchange, skewing (unimodular)	A
Strip-mining, tiling	D, Θ
Pipelining (multidimensional)	β
Parametric pipelining (multidimensional)	Γ
Reversal (unimodular)	A, Γ
Motion, fusion, fission (distribution)	β
Privatization, contraction	F

Considerable advances in dependence analysis, optimization and parallelization heuristics, and code generation proved that polyhedral model is scalable enough to be used in industrial tools. Yet, the problem of devising the optimal transformation sequences for optimizing locality and enabling parallelism is still a topic of considerable ongoing research efforts.

There are two approaches to optimizing programs using polyhedral model: static analytical modelling and iterative optimization. The former builds an analytical model and tries to statically predict the best possible loop transformation. Iterative optimization takes a feedback-directed approach, building different versions of the input program by applying different optimizations and choosing the one that gives the best performance gains. Those two approaches are complementary: analytical modelling can miss the best optimization opportunity but takes just a one pass to complete. On contrary, iterative optimization might search for the best optimization but the search space might be huge, taking many iterations to complete, a challenge for its adoption in production compilers.

Our approach combines an analytical model and an iterative, feedback-directed approach. We rely on the loop tiling framework of Bondughula et al. [29], and on the associated tool Pluto, to extract blocked/tiled loop nests that exhibit one or more parallel loop levels. This framework includes a heuristic to select the shapes of the tiles, to maximize coarse grain parallelism while minimizing communication. This heuristic happens to behave consistently well on our benchmarks. However, another heuristic is proposed to deal with the combinatorics of loop fusion/distribution, and this one is not robust enough to achieve good performance on a wide variety of benchmarks and on multiple architectures. We thus replaced the fusion/distribution heuristic with an iterative search approach, building a search space of feasible, unique transformations before looking for a proper tiling scheme with Bondughula's heuristic. The iterative search is adapted from the more general technique of Pouchet et al. [10], with new invariants enabling to focus the search space on more relevant transformations (for the practicality of the iterative search). Our results on 2 UTDSP and 2 BLAS2 benchmark are reported in Figs. 14, 15, 16, 17, considering two different quad-core general-purpose processors, Intel Core 2 Duo Q6600 and AMD Phenom 9850. Those results demonstrate (1) the potential of our loop-nest optimizer compared to the state-of-the-art (Intel's compiler), and (2) the wide performance differences among the different transformation sequences explored by the iterative search method.

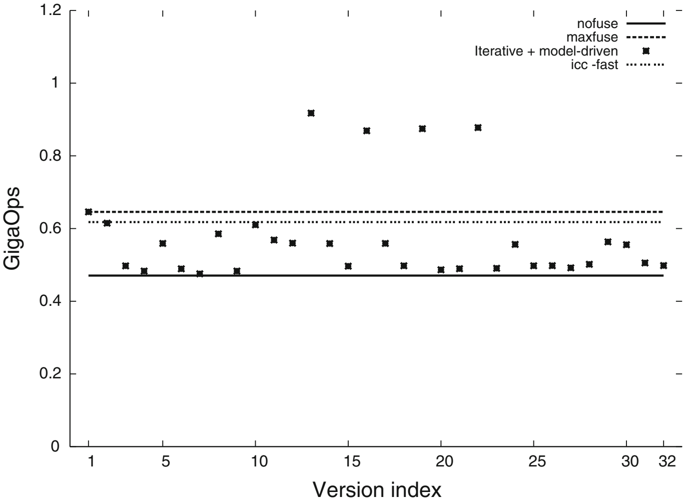


Fig. 14 RegDetect on Core 2 Duo

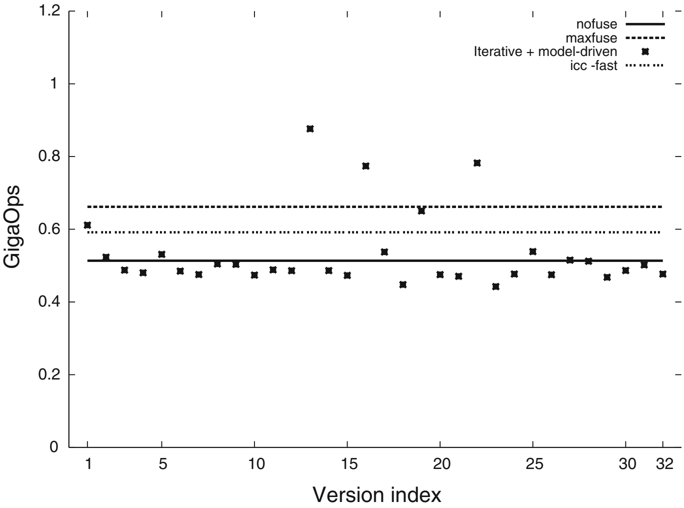


Fig. 15 RegDetect on Phenom

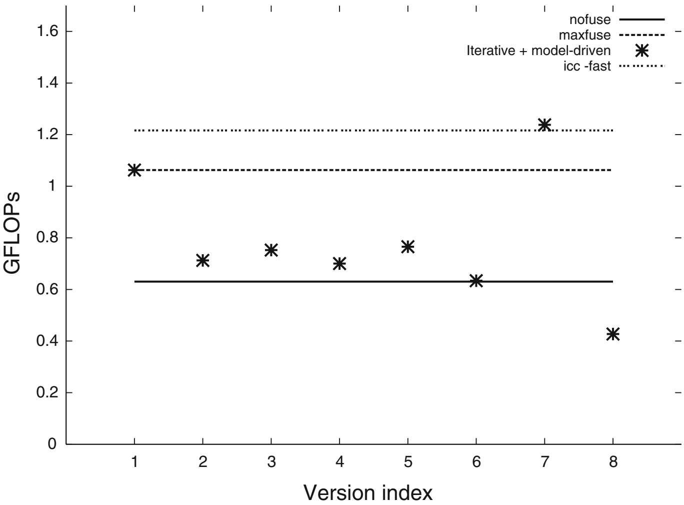


Fig. 16 GEMVER on Core 2 Duo

5.2 Vectorization

Vectorization is the process of converting scalar source code into code that operates on vectors of data elements, making use of SIMD (vector) instructions. Automatic vectorization by a compiler typically focuses on loops, where occurrences of an instruction across different loop iterations operate on different data elements. It can be seen as a downstream stage of the loop nest optimizer, where the selection of target-dependent instructions and access patterns come into play. Vectorization is known to be one of the most effective ways to exploit fine-grain data-level parallelism, and is especially

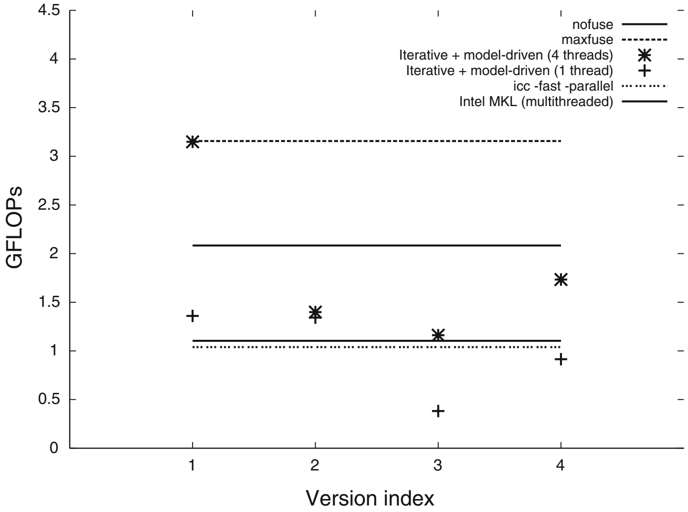


Fig. 17 Doitgen on Core 2 Duo

important for streaming architectures because their processing units typically contain vector (SIMD) units (see Sect. 7) to take advantage of the abundant data-level parallelism available in streaming applications (see Sect. 1.1). Vectorization has therefore been identified as one of the key compiler optimizations to be addressed in the project.

The auto-vectorizer available since GCC 4.0 [30] is capable of vectorizing innermost loops with memory references of unit or power-of-two strides, that may or may not be aligned, and that may include multiple data-types and type conversions, reductions, and other special idioms. The two main enhancements that were missing and were identified as important for the streaming domain are a cost-model, and the capability to vectorize outer-loops. One of the most important goals of the cost model, beyond facilitating informed decision making by the vectorizer, was to be part of an interface to exchange data and guidance with the high-level loop optimizer. This is described in detail in Sect. 5.3. The rest of this section focuses on the in-place outer-loop vectorization capability we developed.

Outer loop vectorization refers to vectorizing a level of a loop nest other than the inner-most, which can be beneficial if the outer loop exhibits greater data-level parallelism and locality than the inner-most loop. Figure 18c shows the result of vectorizing the outer `i` loop of the loop nest in Fig. 18a, assuming Vector Length `VL=4` and `M` divisible by 4. Notice that the innermost `j` loop continues to advance in steps of 1 (compared to 4 in the case of innermost loop vectorization depicted in Fig. 18b), computing 4 results for 4 successive `i` iterations simultaneously.

Outer-loops may have longer iteration counts, smaller strides, more efficient computation constructs, lower initialization and finalization overheads than those in innermost loops, as well as greater potential for promoting vector register reuse, thereby allowing us to leverage the many vector-registers often available in streaming architectures.

```
    for (i=0; i<M; i++){
S1:   s = 0
      for (j=0; j<K; j++){
S2:     s += x[i+j] * c[j]
      }
S3:   y[i] = s
    }
```

(a)

```
for (i=0; i<M; i++){
  vs[0:3] = {0,0,0,0}
  for (vj=0; vj<K; vj+=4){
    vc = c[vj:vj+3]
    vs[0:3] +=
      x[i+vj:i+vj+3] * vc
  }
  y[i] = sum(vs[0:3])
}
```

(b)

```
for (vi=0; vi<M; vi+=4){
  vs[0:3] = {0,0,0,0}
  for (j=0; j<K; j++){
    vc = {c[j],c[j],c[j],c[j]}
    vs[0:3] +=
      x[vi+j:vi+3+j] * vc
  }
  vy[vi:vi+3] = vs[0:3]
}
```

(c)

Fig. 18 FIR-filter Vectorization. **a** Scalar, **b** inner-loop vectorized, and **c** outer-loop vectorized

As mentioned above, data-streaming applications are dominated by nested loops of SIMD-like computations. The high-level data-reuse carried by the outer-loops in these loop nests can be detected and exploited only if operating at the level of the outer-loop. For this reason we have implemented an in-place vectorization approach that directly vectorizes the outer-loop [31–36], instead of the traditional approach of interchanging an outer-loop with the inner-most loop, followed by vectorizing it at the inner-most position [28]. The cost model we developed is capable of guiding the compiler which of these two alternatives is expected to be more profitable (as explained in the following Section).

Operating directly on the level of the outer-loop allows detecting high-level data reuse opportunities that are carried by the outer-loop, as well as fine grained data reuse opportunities related to the handing of alignment. We developed an optimization tapping such opportunities, incorporated within the outer-loop vectorizer. This optimization detects situations in which the misalignment of a vector load in a nested inner-loop is not invariant in the inner-loop (which happens, e.g. if the inner-loop stride S is smaller than the Vector Length (VL)), yet the different misalignments across consecutive inner-loop iterations repeat themselves to form a cyclic group of VL/S distinct misalignments (if S divides VL). This is the case in the example in Fig. 18c where S = 1 and VL = 4. In this case we can achieve fixed misalignment by unrolling the inner-loop by VL/S. Fixed misalignment across iterations can be vectorized much more efficiently, as the misalignment (and in turn, the permutation masks to extract the desired data) can be computed once before the loop (instead of in each iteration). Moreover, each such cyclic group of loads exhibits a high rate of overlap in the data that is being fetched, and can be optimized by removing redundant loads from the unrolled iteration.

Note, however, that such unrolling may result in high register pressure, which on some architectures may result in register spilling, incurring high overhead that masks away the above benefits. For this reason, depending on the required unrolling factor, this optimization may not be suitable for architectures with too few vector registers, but is especially appropriate for streaming architectures, that often include a relatively large number of vector registers (e.g. 128 in the Cell SPE).

We evaluated these techniques on multimedia benchmarks. Our implementation of in-place outer-loop vectorization achieves speedup factors of 2.92× on average across this set of benchmarks, compared to 1.21× achieved by innermost loop vectorization.

Outer-loop vectorization provides superior speedups for most benchmarks due to smaller strides and/or larger outer-loop-counts than those in the inner-loop, and/or by avoiding a reduction-epilog penalty. The optimization for fixed misalignment and data reuse using unrolling is capable of further boosting the performance obtained by outer-loop vectorization, to achieve an average speedup factor of 4.98× (for detailed results see [36]).

These techniques are also applicable to the main computation kernels in the streaming applications described in Sect. 1, namely Gamma-correction and H.264 (in FMradio the inner-most loops are the best choice for vectorization).

5.2.1 Gamma Correction

The complete Gamma-correction algorithm we used for this evaluation is shown in Fig. 19. As the reader will notice, the algorithm continues searching after the interval has been found, but will no longer update the three coefficients. Thus, the algorithm is predictable and more amenable to vectorization. Vectorizing the outer-loop that scans over pixels in a row (the *c* loop), is most effectively done when the inner-most loop (that searches through the array of thresholds, *i* loop in the Figure) is completely unrolled (this means that the pixel-loop becomes the inner-most loop in the nest, and so regular inner-loop vectorization can be applied), achieving an improvement factor of 1.81x/10.15x over the sequential version on PowerPC970/Cell-SPU, respectively. The super-linear speedup on the Cell-SPU (the Vectorization Factor is 4) is due to the heavy penalty for branches on the SPU (which the vectorizer converts into conditional vector

```
volatile int inBuf[LINESZ];  // global arrays
volatile int outBuf[LINESZ];
...
  // Initialize correction coefficients:
  int ta[SN]={...}, oa[SN]={...}, ga[SN]={...};
  for( line=0; line<LINES_PER_FRAME; line++ ) {
    signal_inputFormatter(); // Line buffer empty
    wait_inputFormatter();   // Processor stalled,
    // until line buffer is full.
    for( c=0; c<LINESZ; c++ ) {
      int x = inBuf[c];
      int t=thrh[0], o=offs[0], g=grad[0];
      // Search for correction interval:
      for( i=0; i<SN-1; i++ ) {
        int flag = x>ta[i];
        // Found interval; set pixel coeffs:
        o = flag ? o : oa[i+1];
        g = flag ? g : ga[i+1];
        t = flag ? t : ta[i+1]; }
      // Calculate output:
      outBuf[c] = o + asrrnd((x-t)*g,SCALE );}
    // Line output buffer full; Signal&wait for DMA.
    signal_DMA();
    wait_DMA(); } // Stalled until DMA is ready.
```

Fig. 19 Gamma-correction algorithm

operations) and the data-rotate overhead for operating on scalar data in vector registers (which is avoided when the code is vectorized). Outer-loop vectorization can be used if the inner-most loop is not completely unrolled, e.g. if its loop-count is unknown at compile time or too large to completely unroll. In the particular case at hand, however, the inner-most loop-count ($SN - 1$) is a compile-time known constant 3.

As a point of comparison, we note that these optimizations can be expressed using Silicon Hive's SPM (as described in Sect. 4.6). A manually optimized implementation of Gamma correction for the ISP2300 is shown in Fig. 13. It applies these optimizations (vectorization, loop unrolling) and others (software pipelining with double buffering and explicit synchronization) achieves a 200-fold acceleration, bringing performance up to 0.76 cycles/pixel. Vectorization is expressed using overloaded comparison and conditional operators and by making the pixel loop iterate over vectors of pixels rather than a single pixel, achieving a 30× improvement factor—close to the theoretical speedup factor $VF = 32$. While the data-level parallelism exploited by the compiler using GCC auto-vectorization is comparable to the one achieved manually using Silicon Hive's SPM, the other optimizations that the latter applies are not yet supported by the ACOTES toolchain. In Gamma-correction these further increase data-level parallelism by a factor of 3 on the ISP2300.

5.2.2 H.264

The main computation kernel in H.264 consists of two modes: vertical and horizontal. The first mode has a consecutive access-pattern in the inner-most loop, and an outer-loop that scans through different rows. Here only inner-loop vectorization is applicable, and it obtains a 5.2× to 7.6× performance improvement on PowerPC970/Cell-SPU, respectively. The second mode has a consecutive access-pattern in the outer-loop and a row-size stride in the inner-loop. In-place outer-loop vectorization is applied here and achieves a 10× to 11× performance improvement on PowerPC970/Cell-SPU, respectively. The alternative of first interchanging the two loops in the nest and then applying inner-loop vectorization (rather than vectorizing the outer-loop in-place) achieves a speedup of 6.8× to 11× on PowerPC970/Cell-SPU. In-place outer-loop vectorization is better on PowerPC970 than the interchange based approach due to improved locality (which the Cell-SPU is less sensitive to as it does not have a cache). Reasoning about such tradeoffs and selecting between these alternatives is the role of the cost-model presented in the next section.

5.3 Interaction Between Loop-Nest Optimizations and Vectorization

Vectorization involves low-level, target-specific considerations and transformations, which currently exclude it from being part of the polyhedral framework. In this section, we make a first step in this direction, building a performance model for automatic vectorization integrating seamlessly within the polyhedral representation [37]. Figure 20 summarizes this integration step in the context of the GCC compilation flow. We address a key adaptation problem when porting a streaming application to a new

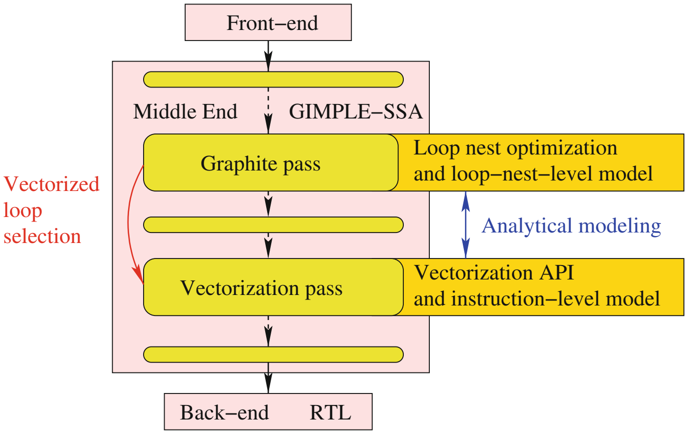


Fig. 20 GCC compilation flow

target architecture; it facilitates educated decision making on how to best apply loop transformations while considering the subsequent effects of vectorization.

To apply vectorization to a loop-nest profitably, an intelligent decision needs to be made as there are often several alternatives to choose from, each with its associated positive or negative performance gain. One such *default* alternative is to keep the original, unvectorized code. Several key costs impact the expected performance of vectorized code, including: strides of accesses to memory, loop trip counts, reduction operations across loops and more. These factors depend on the loop form being vectorized, however, and must be adapted if certain loop transformations are applied to the loop nest. Here we describe how to integrate these factors into the polyhedral model, thereby adapting them seamlessly as different loop transformations are being considered (without explicitly transforming the code), facilitating efficient application of both loop transformations and vectorization.

The underlying assumption of vectorization is that the kernel of a loop usually executes faster if vectorized than if not, but associated overheads may hinder the vectorized version, diminishing its speedup compared to the original scalar version, and more so for loops that iterate a small number of times. Indeed, if the number of iterations N of a loop is smaller than its *vectorization factor* VF, there is no potential for speeding it up using vectorization; on the contrary, vectorizing such a loop may only slow down its execution due to additional preparatory actions and checks. Furthermore, even if N is larger than VF, the number of iterations of the vectorized loop, although positive, may not suffice to out-weigh the overheads incurred by vectorization.

Alongside each transformation we update the cost for individual statements. We can iteratively perform different transformations in the polyhedral model by changing the schedule θ^S for each statement S, computing the cost function of the new schedule. In the end we pick the best possible schedule, based on the minimal cost.

In a classical polyhedral framework, access functions for array references are represented as affine expressions. One may compose this subscript function with static knowledge about the data layout of an array. For each array reference a *linearized* *memory access function* ℓ can capture the stream of memory access locations as a function of the iteration vector:

$$\ell(\mathbf{i}) = b + (\mathrm{L}^{\mathrm{i}}|\mathrm{L}^{\mathrm{g}}|\omega) \times (\mathbf{i}|\mathbf{g}|1)^t = b + \mathrm{L}^{\mathrm{i}}\mathbf{i} + \mathrm{L}^{\mathrm{g}}\mathbf{g} + \omega$$

where b is the base address of the array and $(\mathrm{L}^{\mathrm{i}}|\mathrm{L}^{\mathrm{g}}|\omega)$ is the vector of coefficients that encodes the layout information for data array (assuming row-major data layout). b is typically not known at compilation time; nevertheless, its alignment modulo the vector size is often available from language and `malloc` alignment guarantees. It is crucial for computing the cost of vectorized load/store instructions, which constitutes the majority of the vectorization overhead.

After applying the schedule transformation, the new time-stamp vector is expressed as follows (we ignore β vector component of the schedule for the purpose of data access modelling):

$$\mathbf{s} = (\mathrm{A}|\Gamma) \times (\mathbf{i}|\mathbf{g})^t = \mathrm{A}\mathbf{i} + \Gamma\mathbf{g}.$$

Thus, the original iteration vector $\mathbf{i}$ is given by:

$$\mathbf{i} = \mathrm{A}^{-1}(\mathbf{s} - \Gamma\mathbf{g})$$

which gives us the new, transformed linearized access function

$$\ell'(\mathbf{s}) = b + \mathrm{L}^{\mathrm{i}}\mathrm{A}^{-1}\mathbf{s} + (\mathrm{L}^{\mathrm{g}} - \mathrm{L}^{\mathrm{i}}\mathrm{A}^{-1}\Gamma)\mathbf{g} + \omega$$

with a new vector of coefficients:

$$L' = (\mathrm{L}^{\mathrm{i}}\mathrm{A}^{-1}|\mathrm{L}^{\mathrm{g}} - \mathrm{L}^{\mathrm{i}}\mathrm{A}^{-1}\Gamma|\omega).$$

Thus, linearized access functions (on which the total vectorization cost depends) are transformed automatically with the scheduling transformations. Thus, we do not need to generate code in order to compute the expected vectorization cost after applying a set of loop transformations—the vectorization cost is a function of the scheduling matrix. In the remaining of our presentation we focus on the L^{i} part of the linearized access function coefficient vector.

Our cost model is based on modelling the vectorization cost per statement, given its modified iteration domain $\mathcal{D}^S$ and θ_S. The cost function for statement S is the following:

$$\begin{aligned} c_{\mathrm{vec}}(S, d_{\mathrm{v}}) = \frac{|\mathcal{D}^S|}{VF}\left(\sum c_{\mathrm{vect\_instr}}\right) + \\ \sum_{m\in(\mathcal{W}_S)}\left(f_a + \frac{|\mathcal{D}^S|}{VF}(c_{\mathrm{vect\_store}} + f_m)\right) + \\ \sum_{m\in(\mathcal{R}_S)}\left(f_a + \frac{|\mathcal{D}^S|}{VF}(c_{\mathrm{vect\_load}} + f_s + f_m)\right), \end{aligned}$$

where $|\mathcal{D}^S|$ is the total number of iterations for statement S (taking into the account all nested loops enclosing statement S). VF is the vectorization factor, whereas d_v represents the depth of the loop considered for vectorization. It is implicitly assumed that the cost function depends on the iteration domain $\mathcal{D}^S$ and θ_S (through the change of linearized access function).

Given a loop level d, we can extract the stride δ of a memory access with respect to d by simply looking at element d of vector L^i:

$$\delta_d = \mathrm{L}^\mathrm{i}_d.$$

The stride at the vectorized loop level is obtained by multiplying the relevant element from the linearized access function by VF:

$$\delta_{d_\mathrm{v}} = \mathrm{L}^\mathrm{i}_{d_\mathrm{v}} \cdot \mathrm{VF}.$$

Factor f_s considers the penalty of load instructions[1] accessing memory addresses with a *stride* across the loop being vectorized. Typically, unit-strided (i.e. consecutive) accesses to memory are supported most efficiently by vector loads and stores, incurring minimal if any overhead. However, accesses to non-unit strided addresses may require additional data unpack or pack operations, following or preceding vector load or store instructions, respectively [38]. For example, VF scalar accesses to memory addresses with stride δ_{d_v} across the loop being vectorized may require δ_{d_v} vector loads (each with cost c_1), followed by $\delta_{d_\mathrm{v}} - 1$ vector extract-odd or extract-even instructions (each with cost c_2) to produce one vector holding the desired VF elements. On the other hand, if several accesses to the same address are vectorized together (i.e. $\delta_{d_\mathrm{v}} = 0$), a vector "splat" instruction is often required to propagate the loaded value across all elements of a vector (with cost c_0). Equation 1 shows how factor f_s is computed as a function of the stride δ_{d_v}:

$$f_s = \left\{ \begin{array}{lc} \delta_{d_\mathrm{v}} = 0 : & c_0 \\ \delta_{d_\mathrm{v}} = 1 : & 0 \\ \delta_{d_\mathrm{v}} > 1 : & \delta_{d_\mathrm{v}} \cdot c_1 + (\delta_{d_\mathrm{v}} - 1) \cdot c_2 \end{array} \right\}. \tag{1}$$

Factor f_a considers the *alignment* of loads and stores. Typically, accesses to memory addresses that are aligned on VF-element-boundaries are supported very efficiently, whereas other accesses may require loading two aligned vectors from which the desired unaligned VF elements are extracted (for loading) or inserted (for storing).

This alignment overhead may be reduced considerably if the stride δ of memory addresses accessed across loop levels $d_\mathrm{v} + 1, \ldots, d^S$ is a multiple of VF, because the misalignment remains constant inside the vectorized loop. In this case there is an opportunity to reuse loaded vectors and use invariant extraction masks. By having the transformed linearized access function:

$$\ell(\mathbf{i}) = b + \mathrm{L}^\mathrm{i}_1 i_1 + \cdots + \mathrm{L}^\mathrm{i}_{d_\mathrm{v}} i_{d_\mathrm{v}} + \cdots + \mathrm{L}^\mathrm{i}_{d^S} i_{d^S} + \mathrm{L}^\mathrm{g}\mathbf{g} + \omega$$

[1] Storing vectors with strided access is not yet implemented in GCC.

it is easy to check if misalignment inside the vectorized loop remains constant: coefficients from $\mathrm{L}^{\mathrm{i}}_{d_{\mathrm{v}}+1}$ to $\mathrm{L}^{\mathrm{i}}_{d^S}$ (corresponding to strides of all inner loops of the vectorized loop) must be multiples of VF.

If the misalignment is constant inside the vectorized loop we also check if the base address which is accessed on each first iteration of the vectorized loop (d_{v}) is known to be aligned on VF-element-boundary; if so then there is no need for re-aligning any data: $f_a = 0$. This is done by considering strides across outer-loops (enclosing the vectorized loop, if exist), and initial alignment properties such as array alignment. In order to check the alignment in outer loops, we need to check if coefficients from $\mathrm{L}^{\mathrm{i}}_1$ to $\mathrm{L}^{\mathrm{i}}_{d_{\mathrm{v}}-1}$ are multiples of VF.

By putting together all considerations for alignment, the alignment cost can be modelled as:

$$f_a = \left\{ \begin{array}{lcc} \text{aligned} & : & 0 \\ \text{var. misalign.} & : & |\mathcal{D}^S|(c_1 + c_3 + c_4) \\ \text{fixed misalign.} & : & |\mathcal{D}^S_{1..d_{\mathrm{v}}-1}|(c_1 + c_3) + \\ & & |\mathcal{D}^S|(c_1 + c_4) \end{array} \right\} \tag{2}$$

where c_3 represents the cost of building a mask based on the misalignment amount, c_4 is representing the cost of extraction or insertion and c_1 is the vector load cost. $|\mathcal{D}^S_{1..d_{\mathrm{v}}-1}|$ denotes the number of iterations enclosing the vectorized loop level.

The vectorization factor VF of a loop is determined according to the size of the underlying vector registers and the smallest data-type size of variables appearing inside the loop. Each individual vector register will thus be able to hold VF variables of this small size. However, if there are variables in the loop of larger size, storing VF copies of them will require multiple vector registers, which in turn implies that the associated instructions need to be replicated. Factor f_m records the extra overhead that is associated with this replication.

Additional factors that depend on specific machine resources may also impact the performance of vectorization, such as the size of register files, available ILP, and complex vector instructions.

Taking the kernel in Fig. 18 as an example, the linearized access function for arrays x and c are as follows:

$$\ell_{\mathrm{x}}(\mathbf{i}) = b + (1\ 1 \mid 0\ 0 \mid \omega) \times (\mathbf{i}|\mathbf{g}|1)^t$$
$$\ell_{\mathrm{c}}(\mathbf{i}) = b + (0\ 1 \mid 0\ 0 \mid \omega) \times (\mathbf{i}|\mathbf{g}|1)^t .$$

After performing loop interchange transformation by applying the interchange matrix:

$$\mathrm{A}'^{S_2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

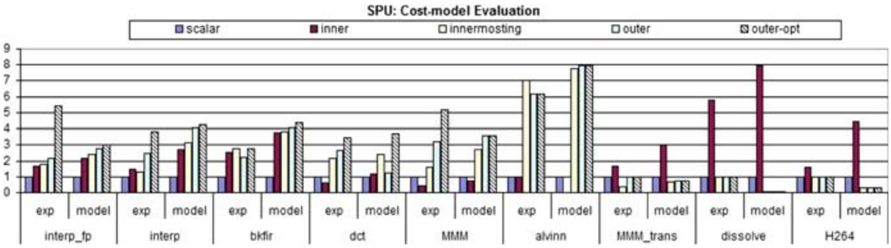


Fig. 21 Cost model evaluation: comparison of predicted and actual impact of vectorization alternatives on the Cell SPU

the new access functions become:

$$\ell'_{\mathtt{x}}(\mathbf{i}) = b + (\,1\ 1\ \big|\ 0\ 0\ \big|\ \omega\,) \times (\mathbf{i}|\mathbf{g}|1)^t$$
$$\ell'_{\mathtt{c}}(\mathbf{i}) = b + (\,1\ 0\ \big|\ 0\ 0\ \big|\ \omega\,) \times (\mathbf{i}|\mathbf{g}|1)^t\ .$$

Notice that the strides have changed. If we choose to vectorize the innermost loop, before the transformation, the access stride in matrix c with respect to the vectorized loop was 1, while after loop interchange the stride with respect to the vectorized loop is 0. The cost function is updated accordingly.

By applying different loop interchange transformations and considering different loops to vectorize, the expected performance varies considerably. Our model is able to predict the best possible combination of loop interchange and outer/inner vectorization strategy.

We evaluated our approach by introducing our model into the polyhedral framework of GCC[2], and comparing its performance estimates for different loop interchanges and vectorization alternatives with actual execution runs of a set of benchmarks. The set of benchmarks includes a rate 2 interpolation (*interp*), block finite impulse response filter (*bkfir*), an 8 × 8 discrete cosine transform for image compression (*dct* [39]), 2D-convolution by 3 × 3 filters for edge detection (*conv*), a kernel from H.264 (*H264*), video image dissolve (*dissolve*), weight-update for neural-nets training (*alvinn*) and a 16 × 16 matrix-matrix multiply (*MMM*) (including a transposed version *MMM_trans*).

Figure 21 displays results on the SPU. In all but one case (alvinn) the model correctly predicted the best vectorization technique. Using the cost-model driven approach, we obtain an average speedup of 3.5x over the scalar version, which is an improvement of 36% over the optimized in-place outer-loop vectorization technique, and 2.3x times better than the innermost vectorization approach, on average.

Figure 22 displays results on the PPC970. The cost model mispredicts in three cases (interp, bkfir and alvinn). The overall speedup obtained by the cost-model driven approach is 2.9x over the scalar version, an improvement of 50% over outer-opt, and 2.3x times better than innermost loop vectorization, on average.

Incorporating the vectorizer into the polyhedral framework is still in progress, and not fully automated yet. Until auto vectorization is fully integrated into the polyhedral

[2] Graphite, http://gcc.gnu.org/wiki/Graphite

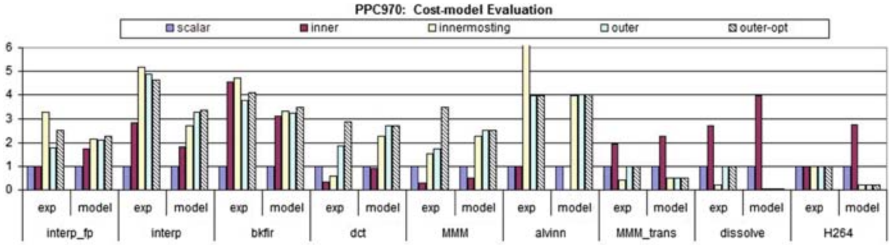


Fig. 22 Cost model evaluation: comparison of predicted and actual impact of vectorization alternatives on PPC970

interface, the vectorizer provides information to the high level loop optimizer via an interface which exports its analysis utilities and allows the high level loop optimizer to query which loops could be vectorized, thereby improving the effectiveness of the high level loop optimization heuristics. This has the potential of speeding up the exploration of loop transformations considerably. The following utilities were identified as useful to assist the high level optimizer in selecting the right loops to optimize for vectorization:

1. given a loop, return whether the loop is vectorizable or not;
2. given a loop, return an iteration count threshold for profitable vectorization;
3. given a loop and an associated number of iterations, return performance impact of vectorizing the loop.

This API is meant to assist the high level loop optimizer when strip-mining or coarsening loops for the auto parallelizer versus the auto vectorizer. It is implemented in GCC, and is available to be used by passes outside vectorization.

6 Compiler Back-End

This section describes the final stage of the ACOTES tool-chain which involves compiler back-ends and code generation for the relevant streaming platforms considered, including the runtime support of these platforms. Previous sections described uniform, largely target-independent stages of the ACOTES tool-chain, from the programming model and programmer annotations through high-level transformations down to low-level transformations using GCC. From this point several alternatives were considered to support code generation for the different streaming targets provided by the partners.

6.1 Common Compiler Backend Format

To increase the mutualization of effort among partners, we studied the suitability of adopting common compiler back-end formats. Several candidates exist for this purpose. A natural one was the GCC RTL format. For one platform, namely the Cell Broadband Engine, the native GCC RTL format was employed. The GNU tool-chain served the development of the Cell processor from its early architecture exploration and programming environment creation [40,41], and GCC already provided the ability to generate code for both PPE (PowerPC Engine) and SPE (Synergistic Processing Engines) cores. Additional enhancements were developed to improve automatic vectorization and other Cell-specific transformations, needed to optimize streaming applications [42].

Alternatively, we considered the suitability of higher level formats, such as the Java bytecode or the Low Level Virtual Machine (LLVM) [43]. Finally, the ECMA-335 Common Language Infrastructure (CLI) standard [44], at the basis of the Microsoft .NET technology, was seriously considered. The motivations were multiple: first of all it is an open standard, providing a clear separation between multiple compiler front-ends producing CLI and multiple CLI consumers, thereby naturally supporting multiple ISA mappings; secondly, it supports the semantics of multiple languages including C, which is used for our project. And finally, thanks to Microsoft, it is a widely adopted technology with growing interest also in the open source community, as indicated by the Mono [45], DotGNU Portable.NET [46] and ILDJIT [47] projects.

The outcome of our investigation [48] is that GCC RTL is indeed a natural candidate for compiler back-end format. For what concerns the high-level processor-independent formats, the characteristics of CLI bytecode and its data representation make it an excellent choice for ACOTES: the framework has been standardized by two different entities: ECMA and ISO, and is stable. It can be used as a processor independent format or can be specialized for a particular target. Additional information can be added to the code, variables and types to drive further optimizations. Finally it supports both managed and unmanaged environments.

In order to exploit the GCC4-based developments of other partners and share this format within the project, STMicroelectronics developed a GCC4 to CLI translator [49,50]. We are leveraging existing efforts in this direction by the open source community, consolidating and complementing their work by adding the streaming information. This translator is free software.

6.2 Split Compilation

Split compilation is a special combination of multi-staged and deferred compilation, where the optimization process is decomposed into *collaborating stages*. In our context, the first stage occurs on the developer's workstation; it takes C code as input and generates a program representation in CLI format. The second step occurs on the device where the application should run. It reads the CLI format and generates the native binary format. This second step can take place at *install-time* or *run-time*, depending on the system.

We take advantage of this two-stage process to transfer the complexity of optimization algorithms towards the first stage, while retaining the ability to specialize the code to the target architecture and execution context. When an optimization cannot be applied in the second stage—either because it is target-dependent, or because it may increase code size too much, or because it is too costly to be applied at runtime—it might still be considered: an offline analysis can build a characterization of the validity and profitability of this optimization, and encode its results into annotations embedded in the intermediate format. The second stage can rely on this annotations, skipping expensive analysis to implement straightforward code transformations. Annotations may also express the hardware requirements or characteristics of a piece of code (I/O required, benefits from hardware floating point support, etc.).

Using this multi-stage process, we can apply the most aggressive techniques like iterative optimization [51] or transformation in the polyhedral model [27] to embedded compilation.

7 Target Streaming Platforms

The ACOTES project targets four streaming platforms. Among these, two (the Cell/B.E. and the HiveFlex ISP2300) are in production, while the other two (the exSTream and the Ne-XVP) are in the design stage.

7.1 Cell B.E. from IBM

The Cell/B.E. [52] is a heterogeneous multicore processor that consists of an IBM 64-bit Power Architecture core, called the IBM PowerPC processor element (PPE), augmented by eight specialized single-instruction multiple-data (SIMD) coprocessors (See Fig. 5a). These coprocessors, called synergistic processor elements (SPEs) [53], provide data-intensive processing; they operate from a local storage that contains instructions and data for a single SPE; this is the only memory directly accessible from the SPE. Memory access is performed via a DMA interface using copy-in/copy-out semantics.

The PPE is fully compliant with the 64-bit PowerPC Architecture and can run 32-bit and 64-bit operating systems and applications. The SPEs are independent processors, each running its own individual application programs that are loaded by the application that runs on the PPE. The SPEs depend on the PPE to run the operating system, and, in many cases, the top-level control thread of an application. The PPE depends on the SPEs to provide the bulk of the application performance. The SPEs are designed to be programmed in high-level languages and support a rich instruction set that includes extensive single-instruction, multiple-data (SIMD) functionality. However, just like conventional processors with SIMD extensions, use of SIMD data types is preferred, not mandatory.

The PPE and the SPEs are compiled separately by different back-end compilers and then linked together to compose an application. CELL/B.E. introduces multilevel parallelism which users must exploit to gain the best performance. The lower level is the SPE level in which vectorization of inner and outer loops should be explored. The higher level is functional parallelism level that should enable to distribute the processing between several SPEs.

A first step towards enabling auto-vectorization for the Cell is to model the SPE registers and instructions as vector registers and instructions in the compiler. In GCC this is done in special machine-description files in the GCC port for the Cell SPE.

Once this model is in place the existing auto-vectorization capabilities of the compiler are transparently enabled.

7.2 Ne-XVP Architecture from NXP Semiconductors

The Ne-XVP architecture (Nexperia eXtreme Video, or Versatile Processor) from NXP Semiconductors Research provides high silicon efficiency and low power for streaming applications like video processing (video encoding and decoding, frame rate conversion, image improvement), while keeping a high level of adaptation to algorithmic change and application diversity. It is therefore a scalable architecture based on duplication of cores (programmable or not) sharing a common memory structure. A careful selection of core characteristics achieve high efficiency for each application domain while keeping a low cost of ownership by reducing the verification and validation costs, and reusing common module elements.

The *multi-core* approach is well represented now in the industry: the clock race has reached its limits and the era of multi-core is coming. But contrary to the typical approach for multicores (heterogeneous), the Ne-XVP architecture uses in a better way the available silicon, starting from the observation that existing cores are optimized for single core environment, and not for multicore. An optimum is better reached when the optimization is done globally than by adding separately optimized elements (at least for non linear systems, which is typically the case of the Design Space of architectures). But the elementary cores should of course be very efficient for the application domain. It is the main motivation for using a VLIW architecture, well suited for embedded applications, and a major architecture which was optimized for years for video processing at Philips then NXP: the TriMedia ISA.

At the core level, Ne-XVP does not provide a new microarchitecture from the TriMedia ISA, it only configures the standard architecture parameters such as the number of issue slots, the number of registers, the composition and number of functional units and the size of data and instruction caches. The main change is the support for multithreading in cores, by adding extra registers banks and minimum logics, allowing a fast context switch in a core. This enables a single physical core to appear like several cores from a software. The motivating for adding multi-threading to cores is twofold:

- It eased software development and porting of applications: a multi-threaded application can work either in different cores, or on a single core, so the number of threads is virtualized. A multi-threaded application can work on different instances of the Ne-XVP architecture that physically differs by their number of cores.
- Multithreading makes each core more insensitive to latency: if a thread is waiting due to a cache miss or latency due to a functional unit, then another task might kick in, increasing the use of the core. Even if multithreading adds complexity to the cores, it allows also to decrease the requirements of some part of the cores (latency, bypass, etc), globally adding a very small area.

To reach the high level of Mops/W, the Ne-XVP will then use the maximum parallelism, but not all parts of applications can be parallelized. This is the well-known *Amdhal's law*: it is the irremovable sequential part of the application that will limit the ultimate performance of the architecture on a particular application. Therefore, the architecture contains a part that runs fast to execute the sequential part of the application.

This fact leads to a heterogeneous approach where a core is optimized for sequential tasks while others can work as a *pool* for parallel operations. It should be noted that increasing the different type of cores beyond two did not really bring further improvement. More details can be found in Hoogerbrugge and Terechko [54].

The specialization and efficiency of the types of core can also be increased by adding specific instructions (*custom-ops* in TriMedia terminology). The instructions can of course be implemented in a specific functional unit in the cores (if they are simple), or be shared by different cores, leading to embedding specific coprocessors in the array of cores, depending on the complexity and granularity of the function. This approach is taken in the Ne-XVP approach for the CABAC coprocessor and for coprocessors helping the task managements and memory coherency. From the point of view of cost of development, complex functions can be shared (a process known as co-joining) and therefore integrated as specific coprocessor, because it will not impact the design and verification of each core, which is always tricky and expensive.

The resources are also tuned to the characteristics of the application domains. For example, the coherence is only required at certain particular points (communication between tasks, end of tasks, etc.), which allows to implement a coherence mechanism that has a very light footprint.

To ease the scalability to various video picture sizes, data partitioning is often used to parallelize applications. The hardware task scheduler accelerator takes benefit of this and dispatches the tasks to the available cores, allowing the same binary code to be executed without modification on different instances of the Ne-XVP architecture with various numbers of cores. Data partitioning also permits the *co-joining* of instruction caches, reducing the silicon footprint of the system.

All these considerations led to the Ne-XVP architecture template in Fig. 23. The architecture is composed of several cores of two types: core1 and core2. The first type optimized for sequential code execution is instantiated only once to address Amdahl law's bottleneck, whereas the second type is replicated many times to carry out the parallel workload. Each core, multithreaded, has a data and instruction cache, where the latter can be shared among several cores. Hardware Coherence Coprocessors maintain data cache coherence in hardware, which also governs the cache2cache tunnels. The Synchronization unit features various hardware accelerators (Hardware Synchronization Unit, Hardware Task Scheduler [55], Task Scheduling Unit) for fast inter-task synchronization and communication, which may be deployed independently of each other.

Compared to a *simple* multicore system based on standard "off-the-shelf" TriMedia TM3270 with 128 KB data cache, 64 KB instruction cache, 5 issue slots, 128 registers, without multithreading and coprocessors, the Ne-XVP architecture is much more efficient in silicon area (and power) for applications such as H.264 decoding. The Ne-XVP architecture is very versatile in terms of its programming model. The software can see the Ne-XVP hardware as being:

- A common address space architecture. The Hardware Coherence Coprocessor ensures the coherence of the various versions of the data in different caches.

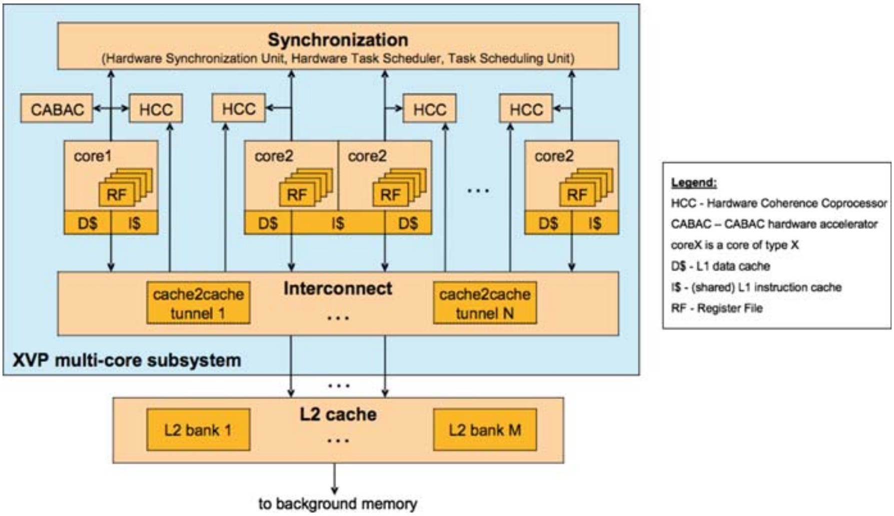


Fig. 23 Ne-XVP architecture

- A distributed memory architecture. Physically, the L1 caches are distributed, and cache-to-cache communication is done by the configurable tunnels.
- A data driven model: the Hardware Task Scheduler allows to activate the various threads running on the cores only when data are ready and available.
- A multi-threaded architecture: the HTS and the cores allow implementing a model similar to the "Cell superscalar" (CellSs).
- A streaming architecture: the typical *push, pop, peek, sneak* functions can be efficiently emulated with the standard Ne-XVP instructions.

Several *streaming* applications were used in the definition of the Ne-XVP architecture; more particularly image processing functions, codec functions (H.264 at standard and Super-HD resolution), image enhancement and 3D Graphics.

For streaming applications mainly composed of a succession of kernels, Ne-XVP can be nearly as efficient as dedicated hardware: each kernel can be mapped on a core, and each core can be self-sufficient, running only with its internal local memory and registers. For the more complex applications, like H.264 super-HD, Ne-XVP allows to easily implement various thread level parallelisms. It could be semi-static, where the compiler does not have to guess the dependencies—they are checked at run-time by the Hardware Task Scheduler. The experiments show that performance is increasing quasi proportionally with the number of cores. The programming model is facilitated by the shared memory architecture, which makes it easier for programmers familiar with *C* or *C*++ programming language.

For the ACOTES compiler chain, its various elements are exercised to use the Ne-XVP architecture efficiently: the SMP allows dispatching tasks on different cores (or hardware threads in a core). The communication scheme of ACOTES allows to exploit the efficiency of the Hardware Coherence Coprocessor, requesting coherence only when it is required. Correct placement (linked to the ASM) allows minimizing the inter-core communication, keeping the tunnel units busy. Finally, the cores, based on the TriMedia VLIW Instruction Set Architecture, benefit from the loop and vector processing improvement of the ACOTES GCC compiler chain. The ACOlib library can also use the hardware mechanisms implemented in the various coprocessors to efficiently support the application.

7.3 xSTream Architecture from STMicroelectronics

The STMicroelectronics xSTream architecture is based on the convergence of communication and computing as a way to solve scalability and programmability of high-performance embedded functionalities, such as graphics, multimedia and radio subsystems. In addition it addresses some of the increasingly challenging design and silicon fabrication issues at the architecture, micro-architecture and design levels through the use of advanced techniques such as voltage and frequency scaling (via a globally asynchronous locally synchronous model, GALS), local clock generators adapted to local silicon process variations (to increase yield and fault tolerance), skew insensitive design (via mesochronous and delay insensitive network on chip links) and use of regular cell design flows (for silicon manufacturability).

Figure 24 illustrates a high-level view of a complete system embedding an instance of the xSTream processor fabric.

The system is composed of the traditional "host processing" part on the left side, which we have depicted as a symmetric multiprocessing (SMP) subsystem for future scalability, while the entity on the top-right end of the picture above is the 'streaming engine' of the xSTream architecture. It is meant to be addressing the needs of

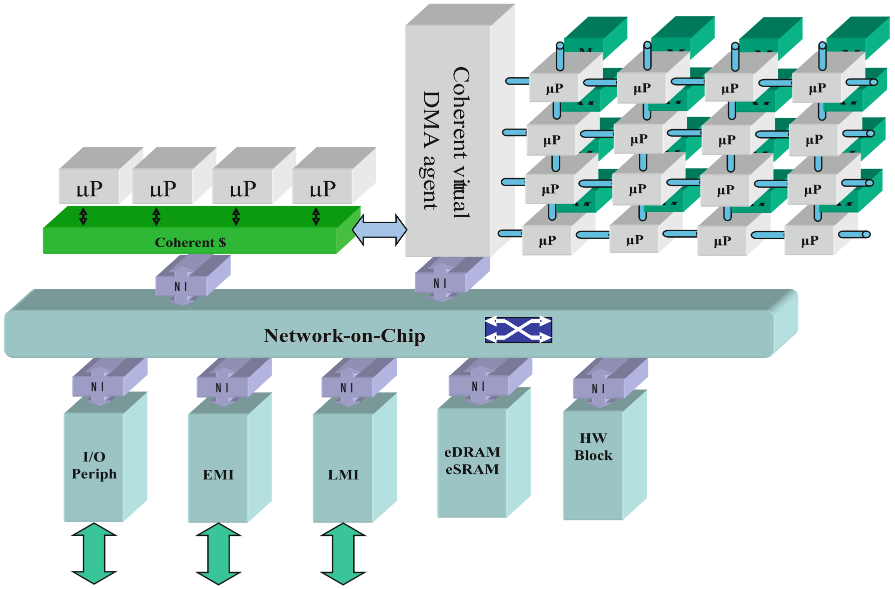


Fig. 24 High-level view of an embedded system containing an xSTream computing fabric

data-flow dominated, highly computational intensive semi-regular tasks, typical of many embedded products. The streaming nature of the kernels mapped onto it makes it possible to design a semi-regular fabric of programmable engines interconnected via a relatively simple network of point to point channels. The tasks structure that is mapped onto the computational fabric is more similar to a pipeline of 'filters' rather than a set of tasks explicitly communicating amongst themselves, while the latter model is not ruled-out by the xSTream template, including up to more traditional lock-based synchronization distributed parallel tasks. It mostly focuses on providing a range of facilities, both HW and SW, to fully exploit data-flow centric applications and programming models. The fabric supports a number of simultaneous software pipelines running on the processing elements to accommodate complex applications and also provide load balancing and latency hiding capability. A property of the streaming fabric is to support very high internal data bandwidth, throughput and computationally intensive tasks (Fig. 25).

The processing elements of the streaming fabric are relatively simple programmable processors or engines with a general purpose but simple basic ISA that can be extended with SIMD or Vector mode instructions. The engines include a set of features for improving performance and efficiency, such as sub-word parallelism, wide data-paths, simple pipelines, etc. At the same time they execute instructions fetched from local memories instead of caches, a great simplification at the pipeline forefront. Local memory is also used for wide data accesses. The engines are connected between them, and the interconnect functionality plays one of the critical roles in this picture.

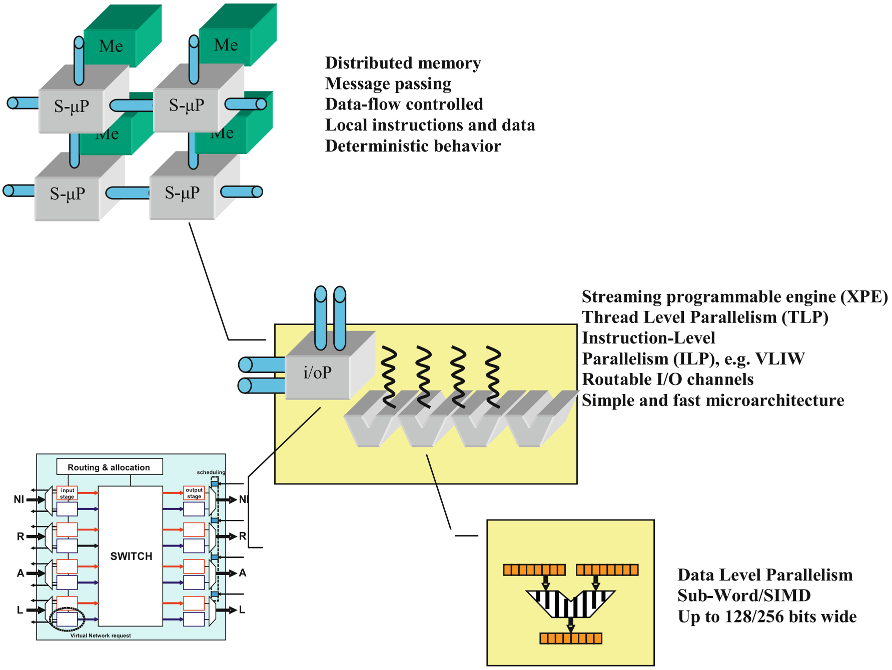


Fig. 25 The xSTream fabric and node structure

In fact it is quite the essence of the system to be able to provide a self synchronizing support for software pipelines. This is achieved with a set of lightweight routers, very similar to the ones being defined for network-on-chip replacements of standard bus infrastructures; but with more freedom for simplification, due to the constrained nature of the communication patterns versus a generic system back-bone NoC. The fabric is not limited to exploitation of programmable engines, in fact it is entirely possible to use hybrid approaches where some elements of the array can be fixed functions, implemented in a classic ASIC design flow, if such function are of a critical, very well known and fixed nature. Likewise the pipelines can be attached on the periphery of it to I/O channels that go to dedicated peripherals or specific I/O functions such as cryptographic engines or similar.

The xSTream architecture provides native support for streaming applications, in particular for what concerns

- communication links,
- communication primitives, which are mapped directly onto native instructions for maximum efficiency,
- memory operations for internal and external memory access,
- a processing element well suited for data intensive computations (the xPE).

The interconnection network of xSTream is composed of a set of routers connected by physical links (for example a 2D mesh topology). Each router has one or more end-points connected to it, which can be a consumer and/or producer of the packets flowing through the network. The end-points can be one of the following: an xSTream accelerator processor or xPE, high-bandwidth IO links to the outside world or DMA channels.

Router-to-router connections as well as router-to-end-point connections are implemented over a single physical channel, whose parameters can be tuned for width, speed, arbitrations, local buffering.

To improve performance and simplify low-level deadlock avoidance, the Network On chip supports virtual channels that are used to multiplex physical connections. The architecture also supports a communication feature at a higher level of abstraction: virtual channels are implemented by using multiple “virtual” queues managed by a Flow Controller (xFC) at each end-point node of the fabric. A packet can be written into a specific “virtual” queue only if the credit based end-to-end flow-control allows it; this guards the application from running into so called high-level application deadlocks that might arise when the data-flow graphs do not satisfy certain conditions. Additionally, virtual queues can be of virtually any length, limited only by the amount of local memory available per each node; this feature greatly extends the freedom of software mapping tools to explore a larger feasible solution space for the mapping problems and objective functions (such as local bottlenecks, load-balancing, etc.), and increases the potential of finding better optimized configurations. Virtual queues at the inputs and outputs of the xSTream xPE processor are exposed at the instruction set level so that they can be effectively treated similarly to fully disambiguated sequential memory references by optimizing compilers.

The following ISA operations are defined for the queues of each xPE and can be described functionally as:

- `pop` destination_reg, queue_identifier
- `push` source_reg, queue_identifier
- `peek` destination_reg, index, queue_identifier
- `qsync` queue_identifier

The queue_identifier is a global queue identifier and specifies the source or destination nodes to which the operation is directed as well as the specific virtual queue on which the operation is to be performed. The `pop` operation retrieves data from the head of the queue and fills the destination_reg. The `push` operation pushes the content of the source_reg into the tail of the queue. In practice queues are defined as variable length in the implementation through a mechanism that uses local memory to the xPE as back-log temporary storage implemented by the xFC.

This same mechanism allows also to support a third primitive, which is the peek operation. The `peek` operation behaves like a `pop` issued after a successive number of `pops` for index-1 elements but does not actually remove the elements from the queue. Effectively it provides partial random access support for the queue itself. A `peek` operation is only allowed for a queue with back-log storage in local memory that has been declared to be large enough to contain a number of elements greater or equal to the maximum peek index used for a given input queue. In practice, `peek` blocks if less than index elements are present in the virtual queue. It may also be used for synchronization purposes and block based processing instead of pure streaming. Finally the `qsync` primitive guarantees that an output virtual queue is drained of all data. This instruction is required for pipeline setup and shutdown, but also to allow advanced management of virtual queues and context switches.

The xSTream architecture template can accommodate various kinds of computing nodes from programmable ones to hardwired functions. To complete the template with a suitable programmable element we have designed a highly parametric programmable engine that we call xSTream processing element (xPE).

The xPE is optimized for stream oriented, data-flow dominated, performance demanding computation. The target application range is especially focused on embedded multimedia, telecom, and signal processing applications. Specific requirements that were used for the definition of the engine architecture and microarchitecture were: low silicon area or more exactly, high computing density in terms of MIPS per physical gate.

High computational power with outstanding power figures in term of MIPS/mW, MIPS/MHz, associated with relatively high (for the embedded world) operating frequencies, support for high level programming languages and compiler friendliness, especially for ISA orthogonality when it comes to vector/SIMD execution semantics. Finally the xPE is highly design-time configurable with a wide range of tuning knobs and optional features enabling extensive tradeoffs for area, power, and compute density to adapt the fabric granularity to a specific application domain.

The xPE microarchitecture is a highly streamlined and minimalist core architecture aimed to tune system frequency and limit core size by shaving off most of the complexity required for more general purpose microprocessors and media processors. The xPE execution semantics is VLIW coupled with a modular and scalable design based on configurable VLIW slices.

Local memories and interfaces are optimized for managing and accessing data streams, as well as wide vector instructions operating on packed data words to exploit available DLP. The xPE supports a fine-grained multithreading to exploit task level parallelism and, once again, to ease the data-flow application mapping tasks and to achieve the more conventional latency hiding benefits of multithreading.

Each xPE slice includes two vector integer/floating point general-purpose ALUs, one vector integer/floating point multiplier unit, one vector load/store unit with special stream operations (push/pop to/from queues), independent fetch/issue unit with supporting features to synchronize multiple slices, shared registers for cluster communication and simple branch prediction features. The xPE data-path supports either 32-bit operands or 128-bit packed vector operations (4 × 32-bit or 8 × 16-bit), has an extensible register file with multiple views, as 64 general-purpose, 32-bit wide registers and one with 32 general-purpose, 128-bit wide vector registers. The xPE pipeline has nine stages with up to four stages of execution depending on functional units' latencies. Full bypassing and forwarding is supported. The vector extensions ISA include three source operand operations for SIMD most of which are capable of a subword permutation operand for greater flexibility to support automated vectorization (Fig. 26).

The xPE ISA is designed for maximum flexibility for further extensions, but with code density very much in mind in terms of efficiency as most of the code will be fetched from relatively small local memory. A short summary of the ISA features is:

- Integer, fixed point, and floating point operations.
- Special instructions for thread synchronization.
- Instructions are encoded with 24-bit syllables.
- Up to 2 syllables, packed in a bundle, may execute in parallel.
- Wide immediate values and various bundle extension formats.
- Variable bundle size to optimize code size.

7.4 Silicon Hive HiveFlex ISP2300 Subsystem Architecture

Silicon Hive's basic system-level processing template is an IP-based structure containing an arbitrary number of cores, interconnected through buses, point-to-point connections, and streaming connections. Figure 27 depicts the template for a single processor. The processors within a system may all be different. Each may use various amounts of different types of parallelism: sub-operation-level parallelism (pipelining within operations), operation-level parallelism (domain specific operations with many inputs and outputs), data-level parallelism (vector, SIMD), and instruction-level parallelism (ILP, VLIW). Using this template, Silicon Hive develops domain-specific multi-processor subsystems for streaming applications.

The ACOTES project targets streaming applications and has chosen three applications from three different streaming domains: H.264 video processing, FM radio modulation (communication), and Gamma correction (image processing). It was found that each of these domains exhibit different kinds of parallelism and thus require different processor architectures. A communications processor needs to rely more on ILP. Image signal processing is more regular and thus can benefit most from extensive use

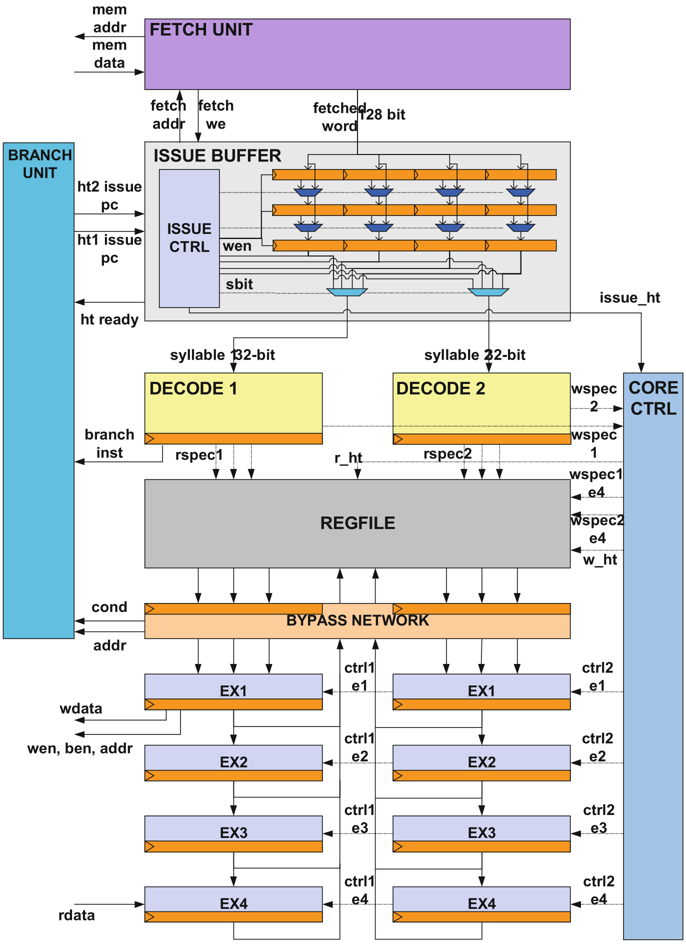


Fig. 26 The xPE slice data-path

of vectorisation. Lastly, video processing is a mix of control and image processing. Thus, a video processor may consist of multiple smaller vector processors, combined with a scalar control processor.

Next to applying the template features, as described above, the processors themselves need to be scalable, in terms of the above architectural parameters. Before committing a processor design to silicon, the architectural parameters are fixed. For example, the required performance points for a communications processor are obtained by scaling the number of issue slot clusters (ranging from 5 to 20 issue slots for complex arithmetic). The image signal processor (ISP2300) has a fixed set of eight vector issue slots, but its vector and element sizes need to be fixed (typically, they scale from 4 to 128 and from 8 to 16, respectively).

This section discusses HiveGo CSS 31xx camera subsystem in more detail. It contains amongst others a HiveFlex ISP2300 processor, configured with 32-element

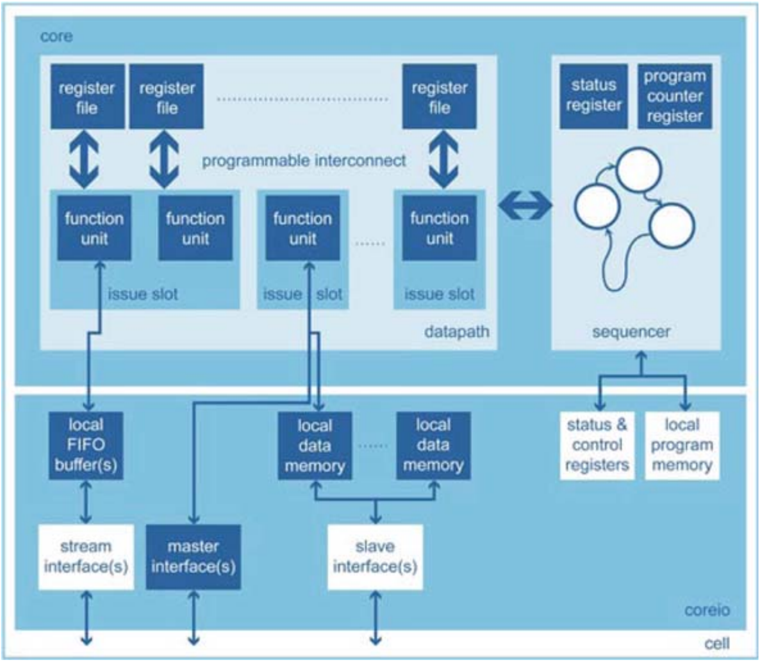


Fig. 27 Silicon Hive processor template

vectors, each element being 16 bits wide. However, the concepts apply equally well to the typical communications or video processors.

Processor subsystems are based on a flexible template; within families, the different processor instantiations also have wide variations in architectural configurations. Thus, as an additional requirement to being able to deal with the different types of parallelism, mentioned above, the software development environment must be able to target a very wide range of different system and processor architectures.

Because of the above wide range of different architectures that need to be supported simultaneously, the requirement is that tools not be changed nor generated to fit the target architecture. Otherwise the number of different tools to be supplied would explode. Thus, each implied tool (e.g. compiler front-end, scheduler, linker, browser, simulator, etc.) must read the system description and target its operation to the system.

Subsystems consist of processors, system-level IP blocks, and stream processing code.

Figure 28 depicts the HiveGo CSS 31xx camera subsystem. In this system, HiveFlex ISP2300 and Scalar Processor are ISO-C programmable, while DMA, InputFormatter, FilterbankAccelerator and GDC/ScalingAccelerator are weakly programmable (autonomous) modules. For the purposes of this paper, they may be considered parallel processors, streaming data into and out of memories inside the ISP2300. Their operation needs to be synchronized with the application running on the ISP2300.

InputFormatter reads raw data coming from the sensor through the SMIA/MIPI interface. The data is packed to fit the vector elements of ISP2300. ISP2300 itself

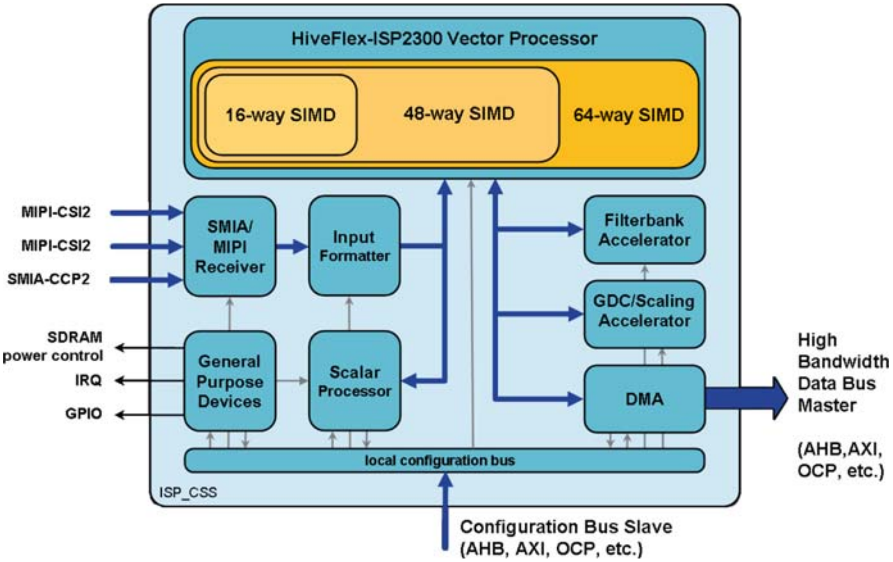


Fig. 28 Block diagram of HiveGo CSS 31xx subsystem

performs a chain of operations on each pixel, whereby the raw sensor data is converted from a Bayer format into weighed RGB pixels. In addition to that, typical camera operations, such as white balance correction and Gamma correction are performed. For typical convolutional filtering (motion blur, etc.) and scaling, the software architect may program the operation of the associated parallel processors to fit between certain software loops. See Fig. 3, where co-processor-implemented functions, such as GDC/Upscaling phase, may be inserted between the color enhancement and sharpness enhancement phases, and where certain image enhancement phases would typically be executed on the Filterbank block.

8 Tool-Chain Evaluation

In this section we demonstrate how the concepts, tools and optimizations presented thus far can be used together in one vertical toolchain. We use the FMradio application (see Sect. 1) as an example, and show how each step in the overall ACOTES framework can be applied to it.

The machine used for evaluation is a 4-core Power6 with 2-way SMT in each core, 64 KB L1 cache (cache line size 128B) and 2 MB L2 cache per core, running under Linux. Experiments were done using all eight hardware threads (i.e. with affinity set to “all”).

8.1 Using the SPM

The parallelism in FMradio can be exposed using the ACOTES directives and Front-End tools described in Sects. 4.1–4.3 (version from now on denoted “SPM”) or

```
taskgroup
{
  Reader (&pair))

  for (i = 0; i < 8; i++) {
    task input (pair) output (fm_qd_bp)
      FM_QD_Demod (pair, &fm_qd_bp)
    task input (fm_qd_bp) output (band_11)
      FFD (1.813, fm_qd_bp, &band_11
    task input(fm_qd_bp) output(band_12)
      FFD (1.813, fm_qd_bp, &band_12
    task input(fm_qd_bp) output(band_21)
      FFD (1.407, fm_qd_bp, &band_21)
    task input(fm_qd_bp) output(band_22)
      FFD (1.407, fm_qd_bp, &band_22)
    task input(band_11,band_12,band_21,band_22) output(ffd_bp)
      subMultSq (band_11,band_12,band_21,band_22, &ffd_bp)
  }
  task input(fm_qd_bp) output (band_2)
    FFD (8.407, fm_qd_bp, &band_2)
  task input(ffd_bp) output (band_3)
    FFD (8.407, ffd_bp, &band_3)
  task input(band_2,band_3)
  {
    stereo_sum (band_2, band_3, &output)
    Writer (output)
  }
}
```

Fig. 29 Annotated FMradio

using standard OpenMP pragmas, with the minor additional extension described in Sect. 4.4 (further denoted "GCC-SPM"). The GCC-SPM's implementation, in the "streamOMP" branch of GCC, is still ongoing, so we will provide results based on manual streamization for this version. We demonstrate and evaluate both approaches.

We observed that 13 tasks can be distinguished and linked using streams. Of such tasks, five of them are very light, and we have joined them in two sets of three and two tasks. This leaves a total of ten exploitable tasks. We have annotated the code with the ACOTES directives, compiled it with the ACOTES compiler, and linked using ACOlib. Figure 29 shows the structure of the resulting code. Observe how it describes the same structure as presented in Fig. 1.

The alternative OpenMP-based implementation of the FMradio code presented in Fig. 29 only requires minute modifications, like replacing the `input` and `output` clauses by their OpenMP counterparts `firstprivate` and `lastprivate` and adding the appropriate `parallel` and `single` directives. Figure 30 shows the resulting OpenMP annotated code. Here we only rely on GCC, with OpenMP enabled through the `-fopenmp` compiler option.

We evaluate the impact of streamization using the above two approaches on Power6. The sequential code is identical in both cases. The `SPM` streamized version produced only 1.1x speedup factor over the sequential code because of degraded cache behavior and synchronization overhead. See Fig. 31 for a detailed comparison.

```
#pragma omp parallel
#pragma omp single
{
  Reader(&pair);
  for (i = 0; i < 8; i++) {
#pragma omp task input(pair) output(fm_qd_bp)
    FM_QD_Demod(pair, &fm_qd_bp);
#pragma omp task input(fm_qd_bp) output(band_11)
    FFD(1.813, fm_qd_bp, &band_11);
#pragma omp task input(fm_qd_bp) output(band_12)
    FFD(1.813, fm_qd_bp, &band_12);
#pragma omp task input(fm_qd_bp) output(band_21)
    FFD(1.407, fm_qd_bp, &band_21);
#pragma omp task input(fm_qd_bp) output(band_22)
    FFD(1.407, fm_qd_bp, &band_22);
#pragma omp task input(band_11,band_12,band_21,\
                               band_22) output(ffd_bp)
    subMultSq(band_11,band_12,band_21,band_22,&ffd_bp);
  }
#pragma omp task input(fm_qd_bp) output (band_2)
  FFD(8.407, fm_qd_bp, &band_2);
#pragma omp task input(ffd_bp) output (band_3)
  FFD(8.407, ffd_bp, &band_3);
#pragma omp task input(band_2,band_3)
  {
    stereo_sum(band_2, band_3, &output);
    Writer(output);
  }
}
```

Fig. 30 OpenMP version

Fig. 31 Speedup results obtained from FMradio

Version	Only Stream.	Only Vect.	Stream. + Vect.
SPM	*1.1*	*2*	*1.4*
GCC–SPM	*2.6*	*2*	*3.6*

However, in the case of the GCC-SPM version, these problems were fixed by an optimization [25] implemented in GCC's OpenMP runtime library, libGOMP, whereby the allowed patterns of accesses to streams preclude false sharing of cache lines between producers and consumers. More specifically, this optimization consists of increasing the granularity of accesses to streams by aggregating the reading (resp. writing) of multiple elements in read (resp. write) windows. The size of such windows is a multiple of the size of a cache line, which ensures that producers and consumers never access simultaneously the same cache lines. Thanks to this optimization, this version achieves a 2.6x speedup factor over the sequential code.

8.2 Loop-Level Optimization and Vectorization

FMradio does not exhibit enough nested loops to illustrate the need for complex loop transformations. Loop fusion is the only relevant one, but happens to be always compatible with vectorization in this example; it is systematically applied together with task-level fusion. The compiler (GCC) then proceeds to apply auto-vectorization, as one of the final Middle-End optimization passes. The main computation kernel in FMradio is an inner-loop that scans through the input buffer and computes the sum of products with the array of coefficients. The impact of vectorization is 1.3x/1.4x improvement factor over the streamized versions (as described in Sect. 8.1) using "SPM"/"GCC-SPM", respectively). Alignment handling and reduction overhead (to finalize the summation) are the main factors that explain the gap between the theoretical speedup factor from vectorization (4×) and the speedups we observed.

Relative to the sequential (non-streamized) version, the impact of vectorization is higher, achieving a 2× speedup. These speedups are summarized in Fig. 31.

8.3 Code Generation

The streamized and vectorized code proceeds down the compilation flow, reaching the back-end target-dependent compilation passes, all the way through final code generation, using the machine description files (of Power6 in this example), where the Acolib/OpenMP constructs of the "SPM"/"GCC-SPM" are translated to pthreads library calls. The overall impact of streamizing and vectorizing FMradio is 1.4x/3.6x, respectively.

9 Conclusions

In this paper we presented and demonstrated the framework developed by the ACOTES project. ACOTES includes partners from both industry and academia, whose goal is to improve programmer's productivity using: (1) automatic simulation and compilation techniques to abstract the underlying multi-core hardware from the programmer, and (2) programmer hints (pragmas) that define the inputs, outputs and control variables of the computation, hinting to the underlying compilation system where the borders of the components are. The actual components are then built based on an abstract representation of the platform called the abstract streaming machine (ASM). The ASM expresses the processing thread-level and data-level parallelism capabilities available, and in addition communication overhead (processing and delay) between the processors. The automatic compiler transformations then base their parallelism related optimization decisions on the pragmas and the resources needed by each constructed component mapped to each processor. These techniques and tools were demonstrated using the FMradio streaming program, starting from it's programming using the ACOTES pragmas, through it's multiple levels of compilation, all the way to actual execution on a real streaming architecture.

Acknowledgments The ACOTES project was funded by European FP6 grant IST-STREP-034869 from June 2006 until May 2009. We are grateful to our colleages Andre Kaufmann, Peter Wachsmann and Maxim Lobko who participated in the first year of ACOTES on behalf of Nokia Bochum, Germany, and contributed to the results of our project.

References

1. Blossom, E.: GNU Radio: tools for exploring the radio frequency spectrum. Linux J. **122**, 4 (2004)
2. International Organization for Standardization, ISO/IEC JTC1/SC29/WG11: Coding of Moving Pictures and Audio, Overview of the MPEG-4 Standard, http://www.chiariglione.org/mpeg
3. Balart, J., Duran, A., Gonzàlez, M., Martorell, X., Ayguadé, E., Labarta, J.: Nanos mercurium: a research compiler for OpenMP. In: Proceedings of the European Workshop on OpenMP 2004, October 2004
4. StreamIt Language Specification Version 2.1 (September 2006)
5. Gordon, M.I., Thies, W., Amarasinghe, S.: Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 151–162 (2006)
6. Kudlur, M., Mahlke, S.: Orchestrating the execution of stream programs on multicore platforms. In: Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI'08), pp. 114–124. ACM, New York (June 2008)
7. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. Comput. **36**(1), 24–25 (1987)
8. Feautrier, P.: Scalable and modular scheduling. In: Pimentel, A.D., Vassiliadis, S. Computer Systems: Architectures, Modeling and Simulation (SAMOS'04), number 3133 in LNCS, pp. 433–442. Springer, Berlin (2004)
9. Pop, A., Pop, S.: A proposal for lastprivate clause on OpenMP task pragma, Technical report, MINES ParisTech, CRI—Centre de Recherche en Informatique, Mathématiques et Systèmes, 35 rue St Honoré 77305 Fontainebleau-Cedex, France. http://www.cri.ensmp.fr/classement/doc/A-403.pdf (2009)
10. Pouchet, L.-N., Bastoul, C., Cohen, A., Cavazos, J.: Iterative optimization in the polyhedral model: part II, multidimensional time. In: ACM Conference on Programming Language Design and Implementation (PLDI'08), Tucson, Arizona (June 2008)
11. Fatahalian, K., Horn, D.R., Knightd, T.J., Leem, L., Houston, M., Park, J.Y., Erez, M., Ren, M., Aiken, A., Dally, W.J., Hanrahan, P.: Sequoia: programming the memory hierarchy. In: ACM/IEEE conference on Supercomputing (SC'06) (2006)
12. OpenMP Organization: OpenMP Application Program Interface, v. 3.0. http://www.openmp.org (May 2008)
13. Ayguadé, E., Copty, N., Duran, A., Hoeflinger, J., Lin, Y., Federico, M., Su, E., Unnikrishnan, P., Zhang, G.: A proposal for task parallelism in OpenMP. In: Proceedings of the 3rd International Workshop on OpenMP (IWOMP), pp. 1–12 (2007)
14. Gonzàlez, M., Ayguadé, E., Martorell, X., Labarta, J.: Complex pipelined executions in OpenMP parallel applications. In: Proceedings of the 2001 International Conference on Parallel Processing (ICPP), pp. 295–304. IEEE Computer Society, Washington, DC (2001)
15. Gonzalez, M., Ayguade, E., Martorell, X., Labarta, J.: Exploiting pipelined executions in OpenMP. In: Proceedings of the 2003 International Conference on Parallel Processing (ICPP'03), pp. 153–160 (2003)
16. Bellens, P., Perez, J.M., Badia, R.M., Labarta, J.: CellSs: a programming model for the cell BE architecture. In: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (2006)
17. Nijhuis, M., Bos, H., Bal, H., Augonnet, C.: Mapping and synchronizing streaming applications on Cell processors. In: International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'09). Paphos, Cyprus (January 2009)
18. ILOG. CPLEX Math Programming Engine. http://www-304.ibm.com/jct01003c/software/integration/optimization/cplex
19. Lundgren, W.I., Barnes, K.B., Steed, J.W.: Gedae: Auto Coding to a Virtual Machine. www.gedae.com (2004)
20. Carpenter, P., Ramirez, A., Martorell, X., Rodenas, D., Ferrer, R.: Report on Streaming Programming Model and Abstract Streaming Machine Final Version. Deliverable D2.2, IST ACOTES Project (September 2008)
21. Carpenter, P., Rodenas, D., Martorell, X., Ramirez, A., Ayguadé, E.: A streaming machine description and programming model. In: Vassiliadis S., et al. (eds.), Proceedings of the International Symposium on Systems, Architectures, MOdeling and Simulation (SAMOS), Lecture Notes in Computer Science, vol. 4599, pp. 107–116. Springer, Berlin (August 2007)

22. Fursin, G., Cohen, A.: Building a practical iterative interactive compiler. In: 1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), Colocated with HiPEAC 2007 Conference (2007)
23. Girona, S., Labarta, J., Badia, R.M.: Validation of dimemas communication model for MPI collective operations. In: Proceedings of the 7th European PVM/MPI Users' Group Meeting, Lecture Notes In Computer Science, vol. 1908, pp. 39–46. Springer (2000)
24. Carpenter, P.M., Ramirez, A., Ayguade, E.: Buffer sizing for self-timed stream programs on heterogeneous distributed memory multiprocessors. In: High Performance Embedded Architectures and Compilers 5th International Conference, HiPEAC'10, pp. 96–110. Springer (January 2010)
25. Pop, A., Pop, S., Jagasia, H., Sjödin, J., Kelly, P.H.J.: Improving GNU compiler collection infrastructure for streamization. In: Proceedings of the 2008 GCC Developers' Summit, pp. 77–86. http://www.gccsummit.org/2008 (2008)
26. Fellahi, M., Cohen, A.: Software pipelining in nested loops with prolog-epilog merging. In: International Conference on High Performance Embedded Architectures and Compilers (HiPEAC'09), LNCS. Springer, Paphos, Cyprus (January 2009)
27. Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M., Temam, O.: Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. Int. J. Parallel Program. **34**(3), 261–317 (June 2006), special issue on Microgrids
28. Allen, R., Kennedy, K.: Optimizing Compilers for Modern Architectures. Morgan Kaufmann Publishers, Burlington (2001)
29. Bondhugula, U., Hartono, A., Ramanujam, J., Sadayappan, P.: A practical automatic polyhedral parallelization and locality optimization system. In: ACM Conference on Programming Languages Design and Implementation (PLDI'08). Tucson, AZ (June 2008)
30. Naishlos, D.: Autovectorization in GCC. In: Proceedings of the GCC Developers' summit, pp. 105–118. ftp://gcc.gnu.org/pub/gcc/summit/2004/Autovectorization.pdf (June 2004)
31. Scarborough, R.G., Kolsky, H.G.: A vectorizing Fortran compiler. IBM J. Res. Dev. **30**(2), 163–171 (1986)
32. Wolfe, M.: High Performance Compilers for Parallel Computing. Addison Wesley, Reading (1996)
33. Ngo, V.: Parallel loop transformation techniques for vector-based multiprocessor systems. Ph.D. thesis, University of Minnesota (1994)
34. Naishlos, D., Biberstein, M., Ben-David, S., Zaks, A.: Vectorizing for a SIMdD DSP architecture. In: Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES), pp. 2–11 (2003)
35. Nuzman, D., Namolaru, M., Zaks, A., Derby, J.H.: Compiling for an indirect vector register architecture. In: Proceedings of the 5th Conference on Computing Frontiers, pp. 199–208 (2008)
36. Nuzman, D., Zaks, A.: Outer-loop vectorization—revisited for short SIMD architectures. In: Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT), pp. 2–11 (October 2008)
37. Trifunovic, K., Nuzman, D., Cohen, A., Zaks, A., Rosen, I.: Polyhedral-model guided loop-nest auto-vectorization. In: Parallel Architecture and Compilation Techniques (PACT'09). Raleigh (September 2009)
38. Nuzman, D., Rosen, I., Zaks, A.: Auto-vectorization of interleaved data for SIMD. In: Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), pp. 132–143 (June 2006)
39. Lee, C.G.: UTDSP Benchmarks. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html (1998)
40. Gschwind, M., Erb, D., Manning, S., Nutter, M.: An open source environment for cell broadband engine system software. IEEE Comput. **40**(6), 37–47 (2007)
41. Weigand, U.: Porting the GNU tool chain to the cell architecture. In: Proceedings of the GCC Developers' Summit, pp. 185–198. Ottawa, Canada (June 2005)
42. Rosen, I., Elliston, B., Eres, R., Modra, A., Nuzman, D., Weigand, U., Zaks, A., Edelsohn, D.: Compiling effectively for cell B.E. with GCC. In: 14th Workshop on Compilers for Parallel Computing (CPC) (January 2009)
43. Lattner, C., Adve, V.: LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04). Palo Alto, California (March 2004)

44. ECMA International: Rue du Rhône 114, 1204 Geneva, Switzerland. Common Language Infrastructure (CLI) Partitions I to IV, 4th edn. (June 2006)
45. Novell: The Mono Project. http://www.mono-project.com
46. Southern Storm Software, Pty Ltd: DotGNU Project. http://dotgnu.org
47. Campanoni, S., Agosta, G., Reghizzi, S.C.: A parallel dynamic compiler for CIL bytecode. SIGPLAN Not. **43**(4), 11–20 (2008)
48. Cornero, M., Rohou, E., Ornstein, A., Ladelsky, R.: Report on Back-end Formats. Deliverable D5.3, IST ACOTES Project (December 2007)
49. Costa, R., Ornstein, A.C., Rohou, E.: CLI back-end in GCC. In: Proceedings of the GCC Developers' Summit, pp. 111–116 (July 2007)
50. Svelto, G., Ornstein, A., Rohou, E.: A stack-based internal representation for GCC. In: First International Workshop on GCC Research Opportunities (GROW), in Conjunction with HiPEAC 2009, pp. 37–48 (January 2009)
51. Bodin, F., Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P., Rohou, E.: Iterative compilation in a non-linear optimisation space. In: Workshop on Profile and Feedback-Directed Compilation (FDO-1), in conjunction with PACT '98 (October 1998)
52. Pham, D., Asano, S., Bolliger, M., Day, M., Hofstee, H., Johns, C., Kahle, J., Kameyama, A., Keaty, J., Masubuchi, Y., Riley, M., Shippy, D., Stasiak, D., Suzuoki, M., Wang, M., Warnock, J., Weitzel, S., Wendel, D., Yamazaki, T., Yazawa, K.: The design and implementation of a first-generation CELL processor. In: Digest of Technical Papers, Solid-State Circuits Conference (ISSCC), Paper 10.2, pp. 184–185 (February 2005)
53. Flachs, B., Asano, S., Dhong, S., Hotstee, P., Gervais, G., Kim, R., Le, T., Liu, P., Leenstra, J., Liberty, J., Michael, B., Oh, H., Mueller, S., Takahashi, O., Hatakeyama, A., Watanabe, Y., Yano, N.: A streaming processor unit for a CELL processor. In: Digest of Technical Papers, Solid-State Circuits Conference (ISSCC), Paper 7.4, pp. 134–135 (February 2005)
54. Hoogerbrugge, J., Terechko, A.: A multithreaded multicore system for embedded media processing. Trans. High-Perform. Embed. Archit. Compil. **4**(2), 168–187 (2008)
55. Al-Kadi, G., Terechko, A.S.: A hardware task scheduler for embedded video processing. In: Proceedings of the 4th International Conference on High Performance and Embedded Architectures and Compilers (HiPEAC), number 5409 in LNCS, pp. 140–152 (2009)