



FP6 Project IST-034869
Deliverable D3.1

List of candidate function- level optimizations

IST-034869

Public

Project Number	:	IST-034869
Project Title	:	Advanced Compiler Technologies for Embedded Streaming
Deliverable Type	:	List

Deliverable Number	:	D3.1
Title of Deliverable	:	List of candidate function-level optimizations
Nature of Deliverable	:	Public
Internal Document Number/version	:	1.0
Contractual Delivery Date	:	30.11.2006
Actual Delivery Date	:	15.01.2007
WP(s)	:	WP 3 High-level transformation parametrised by ASM
Author(s)/Affiliation	:	Albert Cohen (INRIA)

History

<i>Version</i>	<i>Date</i>	<i>Author</i>	<i>Details</i>
1.0	15.01.2007	Albert Cohen	First delivered version

Keyword list

ASM Abstract Streaming Machine
SPM Streaming Programming Model
WP Work Package

Table of Contents

History	2
Table of Contents	3
0 Introduction.....	4
1 Assumptions	5
1.1 Assumptions About the Architecture	5
1.2 Assumption About the Programming Model.....	5
1.3 Assumptions About the nature and scope of function-level optimizations	6
2 Candidate function-level optimizations.	7
2.1 Automatic parallelization	7
2.2 Optimizations for task-level parallelism	7
2.3 Single-threaded interprocedural optimization	7
3 Important ongoing and future work	8
4 References.....	9

0 Introduction

The goal of the ACOTES project is to substantially improve the programmer's efficiency in developing applications that process massive streams of data on programmable, parallel embedded architectures. The target application areas of ACOTES are video processing and advanced radio communications for consumer systems.

Based on the SPM and the ASM, the work in WP3 (High level transformations parameterised by Abstract Streaming Machine) will partition the streaming application at the functional block level, and also identify and implement optimisation strategies at this level.

This document contains a list of candidate optimisation strategies, together with a description of assumptions on the architecture, the Programming Model, and the nature and scope of the optimizations.

1 Assumptions

1.1 Assumptions About the Architecture

- Globally addressable, distributed memory
- Explicit block transfers (e.g., DMA)
- Fine grain distant accesses possible but costly
- Priority of this task: loosely coupled thread-level parallelism
- But also support tightly coupled fine-grain micro-threads (if present)
- Decoupled address generation, data transfer and computation (if present)
- Exposed, complex and/or heterogeneous interconnection network

1.2 Assumption About the Programming Model

- Productivity features
 - Deterministic and compositional parallelism (Kahn principle)
 - Prefer iterators/clocks to sequential control structures (loops)
 - Automatic parallelization, but allow the expert programmer to (portably) express any scheduling or mapping property
 - Interface with (adapted) essential support libraries (libc) and plain C
- Scalability features
 - Express many levels and sources of parallelism
 - Map control, data, and movements to the right level and resources
 - Portable performance
- Efficiency and predictability features
 - Annotation and checking of real-time and resource constraints
 - Interactive programming scheme with manageable feedback from the compiler
 - Guaranteed low-overhead implementations (support OS-free parallelism)

1.3 Assumptions About the nature and scope of function-level optimizations

- Most task-level parallelism will be expressed through functions, with the probable attachment of clock semantics for deterministic and efficient execution.
- Some of these functions will be seen as processes, with a persistent state and affinity to some spatial partitioning/mapping. Task-level concurrency and communication will be implemented through calling other processes.
- A large part of the automatic parallelization support will address task-level parallelism. We expect the programming model to ease the extraction of this parallelism (alias information, localization of state, predigested privatization and tiling of iterator/clocked control flow).
- Opportunities for interprocedural optimization lie within processes as well as more classical functions not directly linked with the expression of parallelism.
- Pure functions are stateless (the same arguments produce the same results) and little affinity with particular distributed resources (except on heterogeneous targets with HW accelerators). These implement the traditional form of functional abstraction (code reuse).
- Plain C functions, including libc and other essential components, not following any particular guideline/semantics introduced in WP2.
- Function-level optimisations in WP3 will target both kinds of functions, on the largest possible scope available in a given compilation unit. Global optimization of multiple object files is not considered in the project but may definitely be a plus if the ongoing projects in the area become mature enough during the lifetime of ACOTES.

2 Candidate function-level optimizations.

2.1 Automatic parallelization

The main goal is to extract parallelism from state-less or state-full but "well-behaved" functions following WP2-defined programming guidelines. Interprocedural alias analysis, and to a lesser extent, interprocedural array region analysis will be necessary.

The construction of independent processes (tasks) depends on these analyses, as well as their synchronizations/communications and the associated iterator/clock annotations.

Internal state removal and privatization is an important enabling transformation that may expose more parallelism thanks to the calling/instanciation context of these processes/functions.

Notice array dependence analysis will benefit from those but does not directly belong to this level of the compilation flow.

2.2 Optimizations for task-level parallelism

The goal of the programmer and of the automatic parallelization passes should be to maximize the extraction of parallelism. The responsibility of choosing which parallelism to exploit and how is left to a downstream set of compilation passes:

- Coalescing of adjacent tasks in the data-flow graph (e.g., adjacent tasks in a pipeline).
- Specialization (with versioning, instanciation of parameters, control-flow normalization).
- Internal state removal can be seen as an optimization similar to dead-code elimination for parallel tasks.

See StreamIt paper at CASES 2005 and typical ArrayOL optimizations.

Tiling and data-parallel slicing of tasks parameterized by iterators/clocks. This capability would look almost like automatic extraction of parallelism, but rely in fact on more abstract iterator/clock annotations of processes.

2.3 Single-threaded interprocedural optimization

Specialization again, sometimes very aggressive in terms of code size, but also very effective. A challenge for cost models.

Function coalescing as an enabling transformation for further loop or back-end transformations (exposing more locality, vector or instruction-level parallelism). It is more economical than inlining and relies on versioning, like specialization.

All existing interprocedural optimizations in GCC are important in the context of ACOTES, but special care will be taken to update their cost models according to the design of the ASM in WP2.

3 Important ongoing and future work

- Most of the basic support for the abovementioned optimizations exist in GCC. Yet the whole middle-end of the compiler will have to be extended along the lines of a new concurrent intermediate representation, otherwise the scope and effectiveness of the proposed passes will be very limited.
- INRIA and NXP are working on a whitepaper about the design of this internal representation. This document will focus on the iterator/clock annotation for processes, which would provide a common interface for higher-level parallel programming models, and which also seems to be critical to allow to characterize fine-grain task-level parallelism and to optimize parallel programs at this level. The time-frame for this whitepaper is early January 2007.
- A major challenge is to build a cost model for these optimizations, based on the ASM design in WP2, and exposing the interplay with the downstream loop transformations, WP4 and WP5. We will study this issue in greater detail when the first version of the ASM is available.

4 References

Important references for this and the upcoming work on function-level, coarse grain parallelism and global optimization in WP3.

- Space-Time Scheduling of Instruction-Level Parallelism on a Raw Machine
Walter Lee, Rajeev Barua, Matthew Frank, Devabhaktuni Srikrishna, Jonathan Babb, Vivek Sarkar, Saman Amarasinghe
Proceedings of the Eighth International Conference on Architectural Support for Programming Language and Operating Systems (ASPLOS-8)
(<http://www.cag.lcs.mit.edu/commit/papers/98/Spacetime-ASPLOS.pdf>)
- Optimizing Compiler for the CELL Processor
Eichenberger, A.E. Oapos Brien, K. Peng Wu Tong Chen Oden, P.H. Prener, D.A. Shepherd, J.C. Byoungro So Sura, Z. Wang, A. Tao Zhang Peng Zhao Gschwind, M.
14th International Conference on Parallel Architectures and Compilation Techniques, 2005 (PACT 2005)
- Cache Aware Optimization of Stream Programs
Janis Sermulins, William Thies, Rodric Rabbah, Saman Amarasinghe
LCTES'05
(<http://cag.csail.mit.edu/commit/papers/05/sermulins-lctes05.pdf>)
- Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs
Michael I. Gordon, William Thies, and Saman Amarasinghe
ASPLOS'06
(<http://cag.csail.mit.edu/commit/papers/06/gordon-asplos06.pdf>)
- Sequoia: Programming the Memory Hierarchy
Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, Pat Hanrahan
SC2006 November 2006
(http://www.stanford.edu/~merez/sc06_sequoia.pdf)
- Compilation for Explicitly Managed Memory Hierarchies, PPOPP 2007.