ASAFESSS: A Scheduler-driven Adaptive Framework for Extreme Scale Software Stacks

Tom St. John
University of Delaware
stjohn@capsl.udel.edu

Benoît Meister
Reservoir Labs
meister@reservoir.com

Andres Marquez
Pacific Northwest National Laboratory
andres.marquez@pnnl.gov

Joseph B. Manzano
Pacific Northwest National Laboratory
joseph.manzano@pnnl.gov

Guang R. Gao
University of Delaware
ggao@capsl.udel.edu

Xiaoming Li
University of Delaware
xli@ece.udel.edu

ABSTRACT

In a context where the interaction between growingly heterogeneous dynamic hardware and sets of complex applications is intractible to model at compile-time, we acknowledge the need for a formal way to associate runtime parameter domains with software optimizations. A standard definition of - and access protocol to - runtime parameters is also necessary for compilers and application developers to adapt their optimizations to the runtime context. We advocate that the runtime parameters should concisely represent the interaction between software and the underlying hardware.

We present a generic framework for collecting a useful set of runtime parameters and creating adaptive optimizations at all levels of the software stack. Runtime task schedulers can be leveraged advantageously to form run-time context information since they have direct access to the tasks that are running and scheduled. In the presented framework, high-level information is produced at the task granularity by the compiler, under the form of "task types," ensuring low management overhead.

The approach is validated experimentally using a simple representation of the run-time context (which we call "runtime mode") that focuses on communication-to-computation imbalance. The software stack element used for validation is an adaptive data compression engine, which in practice could be part of the application or the runtime.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; E.3 [**Coding and Information Theory**]: Data compaction and compression; F.2.1 [**Numerical Algorithms and Problems**]: Computations on matrices

ADAPT'14 Jan 22-22, 2014, Vienna, Austria
ACM 978-1-4503-2514-1/14/01
http://dx.doi.org/10.1145/2553062.2553063.

General Terms

Algorithms, Performance, Theory

Keywords

Adaptive Optimization, Data Compression

1. INTRODUCTION

Dynamic runtime schedulers have experienced broad success in efficiently executing ill-balanced, dynamic parallel programs on multi- and manycore. In particular, task-graph based runtimes [4, 3, 2, 7] have proven to offer new levels of load-balancing and scaling capabilities.

In extreme scale systems, increased disparity between wire and transistor switching speeds aggravates the need for such schedulers, as new hardware solutions to cope with limited power envelope increase hardware heterogeneity, in their functionality (dark silicon) and efficiency (NTV, DVFS).

Scratchpad-based systems also emerge (as seen in GPUs, Runnemede, Cell, and SoCs) to cope with power constraints and the lack of cache scalability. Programming scratchpads requires the explicit specification of data movement in and out of the scratchpad memory.

Good utilization of scratchpads is achieved when data transfers are coarse. Hence, to some degree, extreme scale programs will be characterized by the execution of coarse data transfers followed by computations sized according to the amount of transferred data, and so on. Needless to say, the efficiency of static optimizations is jeopardized in such variable runtime environments, as their benefit is a direct (possibly negative) function of the runtime environment.

In this paper, we explore the explicit discrimination of tasks according to a *task type* that reflects the utilization of the underlying architecture. We expose task type knowledge to the runtime schedulers, and allow them to toggle the software stack between type-specific modes, enabling and disabling optimizations as a function of the current mode.

In Section 2 we give an overview of the envisioned software architecture and of the task types and modes considered in this paper. We illustrate our adaptive framework with runtime type experimentally in Section 4 using a data compression engine presented in Section 3. Finally, Section 5 covers conclusions and future work.

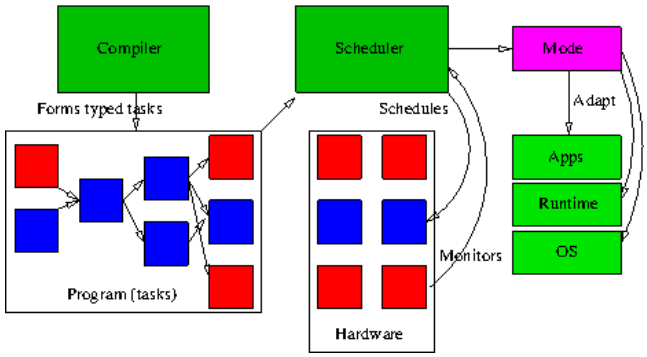

Figure 1: Flow of the scheduler-driven adaptive optimization framework.

2. OVERVIEW

The general idea of this paper is to decompose the execution of programs into tasks of different types, which reflect a quality of the operations within the task. As depicted in Figure 1, the types should be generated as annotations to tasks by the compiler. The runtime scheduler determines a run-time characterization of the set of tasks currently scheduled, which we call "run-time mode." Determining types at the task level enables low mode computation overheads.

Useful modes are the ones to which elements of the software stack (OS, runtime, application code) can adapt. The adaptation model we are interested in aims at optimizing global execution performance and energy efficiency. In this model, the behavior of the elements of the software stack is selected as a function of the run-time mode.

An example of this is a type that would discriminate tasks as a function of their utilization of floating-point operations. A mode that specifies that the currently scheduled tasks are low in floating-point operations would allow a numerical library to use software floating-point emulation in this mode, increasing the profitability of turning floating-point units off. Turning the floating-point units off can be performed by the runtime when the mode represents low utilization. By representing a mode with a vector (of modes) v, this system generalizes to a broad set of optimizations, through the definition of a domain of efficiency D for each optimization o:

$$v \in D(o) \Rightarrow \text{turn } o \text{ to state } x$$

A vector mode seems general enough to handle the runtime optimization of dark silicon processors, in which only a subset of specialized hardware parts (as for instance Taylor's "conservation-cores" [6]) can be powered on simultaneously in order to stay within the processor's power budget.

In this paper we apply this general principle to a simple and universally applicable set of load types, which discriminates computation from communication. We validate our model using an adaptive data compression engine.

3. ADAPTIVE DATA COMPRESSION ENGINE

The optimization considered here is a transformation of a tiled array, in which an array in dense representation may be turned into sparse representation before being processed. Data tiles are marked accordingly as either sparse or dense.

The cost of optimization is roughly linear in the number of elements E in the tile, while the benefit is in $(E-nnz)$ where nnz is the number of non-zeros. Dynamically determining whether a tile is worth compressing by analyzing the data has limited profitability since it is also linear in E. Ideally, we would like compression to occur (mostly) in phases where nnz is small enough and be omitted in phases where nnz is larger.

In terms of execution time, the transformation optimization is worth applying when the computation time for a tile is low as compared to its communication time. Compression is also worth turning off when computations are significantly more important w.r.t. communications (i.e., when nnz is large enough). This defines the two transition rules for our adaptive compression engine, which turns compression on when the commputation-to-communication ratio goes below a certain threshold α and turns compression off when the ratio goes above a different threshold β.

Interestingly, the efficiency domain of the compression optimization is defined in the product of the computation-to-communication ratio and of its own state (on or off).

4. EXPERIMENTAL RESULTS

We have validated our approach using matrix-vector multiplication. To ensure the presence of both sparse and dense blocks in the input matrix, tests cases were generated by modifying the Graph500 [1] generator to produce the initial sparse matrix and overlaying it with dense tiles.

These initial tests were run on four processors using matrices of dimension 2048 and blocks of dimension 128. In these cases, a block was considered to be a suitable candidate for compression if it contained fewer than 128 elements. We observed speedups ranging between 57% and 68% compared to standard matrix-vector multiplication and speedup of 15% compared to sparse matrix-vector multiplication.

5. CONCLUSION AND FUTURE WORK

The presented adaptive framework is general in that it can drive many different software behaviors by modeling them as optimizations associated with an efficiency domain. However, it is limited to optimizations that can make use of the current context. This assumes that the execution of programs happens in *phases* in which properties of the environment don't change too quickly. We plan to explore more such optimizations and to extract a small set of widely-useful task types and modes in the future. We are interested in automating this process by having a compiler introduce type annotations for the scheduler. R-Stream [5] is a natural candidate for this as it natively discriminates communications from computations and forms parallel task-graph programs for SWARM, OCR and CnC. We also intend to quantify phase lengths below which our adaptive framework wouldn't be efficient. Finally, since optimizations impact the environment, we wonder about the collective interaction among a set of adaptive optimizations, and whether using some optimizations jointly could jeopardize the phase execution assumption.

6. ACKNOWLEDGEMENT

This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008717.

7. REFERENCES

[1] The graph500 list. http://www.graph500.org.

[2] Open community runtime. http://01.org/projects/open-community-runtime.

[3] Intel. Concurrent collections. http://software.intel.com/en-us/articles/intel-concurrent-collections-for-cc/.

[4] C. Lauderdale and R. Khan. Towards a codelet-based runtime for exascale computing: position paper. In *Proceedings of the 2nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era*, 2012.

[5] B. Meister, N. Vasilache, D. Wohlford, M. Baskaran, and R. Lethin. R-stream compiler. In *Encyclopedia of Parallel Computing*, pages 1756–1765. 2011.

[6] M. B. Taylor. Is dark silicon useful? harnessing the four horesemen of the coming dark silicon apocalypse. In *Design Automation Conference*, 2012.

[7] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Using a 'codelet' program execution model for exascale machines: position paper. 2011.