Pervasive Parallelism CDT: Automatic parallelization for heterogeneous multi-core architectures with GCC

Chris Cummins · April 13th, 2014

Abstract. In this proposal, I present a research plan to investigate automatic parallelism for heterogeneous multi-core systems. The outcome of this research would be a number of improvements to the automatic parallelization optimiser in GCC. I detail a plan and methodology for achieving these goals, and a review of their significance, and their relevance to existing research of this kind at Edinburgh. The purpose of this document is to be considered for a studentship position in the Pervasive Parallelism CDT under the supervision of Mike O'Boyle and Hugh Leather.

Introduction

It has been well understood that software developers can no longer rely on increasing clock speeds alone to speed up single-threaded applications [1, 2]. While steady performance increases can be gained from exploiting improving CPU cache performance [3], this growth is linear at best, and not the exponential advances that have been afforded in previous years through Moore's Law. Despite this, the adoption of heterogeneous concurrent programming practises has been slow and awkward, due in large part to the clumsy nature of multi-threaded constructs in modern programming languages, and the difficulty and error-prone exercise of programming with them [2]. It is clear that there is an immediate need to address the imbalance between per-core performance and the under-adoption of concurrent programming practises. One such solution is to allow compilers to automatically distribute sequential code into multi-threaded code, relieving the programmer of the responsibility of manual threading by providing a layer of abstraction above the processing units.

It is my hypothesis that the multi-threading of software algorithms will increasingly be considered a facet of resource management. As such, it will be left up to the compiler to manage the allocation and control of threads, in the same way that modern compilers manage the allocation and control of registers and memory in high level languages. Continuing this analogy, it can be observed that the output of quality modern optimizing compilers often outperform the best attempts at handcrafted assembly routines, and so it is imaginable that automatically parallelizing compilers could generate threaded code that would outperform the best handcrafted concurrency. This would bring scalability to existing applications on large core count machines, while simultaneously reducing the difficulty of writing concurrent software.

Background

Automatic parallelization is an advanced feature of optimizing compilers, that to date is often limited to primitive loop parallelization using coarse heuristics [4, 5]. The quintessential example of serial code that can be automatically parallelized is a bounded loop without data dependencies:

```
1 #define N 1000
2 int data[N], i;
3 /* Parallelizable loop: */
4 for (i = 0; i < N; i++)      // Finite bounds, 0 <= i < N
5     data[i] = long_calc(i); // No data dependencies
```

In this case, the loop may be parallelized by assigning different threads for each iteration. The advantage of this parallelism is clear. We can expect to reduce the running time of this loop by a factor of $(n-1)/n$, where n is the number of cores available. However, what is not immediately apparent from this canned example is the increasing importance of parallelizing computation for large core counts. If 95% of a program is parallelized, the 5% of serial code that remains limits the potential speedup of parallelizing to 20x [6]. This does not provide adequate throughput utilisation for the large core counts which are being developed in modern heterogeneous systems, so it is vitally important to achieve parallelization of as much code as is possible.

Existing implementations of automated parallelism have two key areas in which improvements can be made:

1. Heuristics used to decide whether to parallelize code are primitive, using simple profitability metrics of how frequently the code is executed, and whether the number of iterations is large enough to create new threads.

2. The algorithms used to determine whether code may be parallelized do not accurately determine data dependencies, meaning that opportunities to parallelize code are missed.

GCC is a widely used open-source compiler, and has an implementation of automatic parallelization which suffers from the shortcomings identified [7, 8]. In the case of both problems, previous research undertaken at the University of Edinburgh is particularly relevant. While there has been much industry pressure for parallelizing machine learning, no attempts have been made to apply machine learning to automatic parallelization. Mike O'Boyle and Hugh Leather's research into machine learning based compilation offers a great starting point for addressing these issues [9–11]. The feature grammar and genetic algorithms used for automatic feature generation [10] can be adapted to improve the parallelization heuristics, while the self-tuning Milepost GCC offers a practical implementation of a self-tuning compiler which is capable of out performing hardcoded heuristics [11].

Objectives

Based on the existing issues with the GCC automatic parallelization and the research expertise of Mike O'Boyle and Hugh Leather, I have identified a number of objectives as candidate outcomes for research in this area. The objectives have been designed such that real and quantifiable results can be provided for each.

- Improve the quality of parallelization decision logic in GCC using fine-grained statistical heuristics that incorporate thread costs and code size increase in the profitability metrics.

- Apply machine learning and iterative compilation techniques to improve the selection of parallelization optimizations.
- Aggressively increasing the number of loops which can be automatically parallelized in GCC by using dependency reordering for associative operations.
- Develop a technique for employing reusable thread pools to reduce the thread costs of parallelizing operations.
- Implement automatic parallelization of non-innermost loops in GCC using loop collapsing.

Methodology

The first year M.Sc by Research phase would be focused towards background research and developing a finalised project proposal with the help of the supervisors. Supporting courses would include Compiler optimisation, Probabilistic Modelling and Reasoning, and Threaded Programming. The PhD phase of the degree would then be focused upon research and implementation of the finalised project plan.

I am a flexible and self-motivated worker, having experience in both industrial and academic settings of working on projects in large teams or self-directed individual work. My ongoing experience as a regular contributor to open source projects (Linux, GNOME, cogl, clutter) will be extremely beneficial during the practical software development phases, having an intimate knowledge of many open source development processes. By working in a transparent and open source manner, I will provide regular demonstrations and feedback of progress. I have previous experience developing low level software in C in a professional setting at Intel Corporation, providing regular updates to technical employees and higher level progress reports to non-technical management.

Summary

This document proposes a plan for a multi-year research project with both academic and practical implications for the future of optimizing compilers. As the disparity between core counts and computational throughput increases, demand for tested, industry-quality tools will continue to rise, and it is important that there are solutions ready to fit these requirements.

GCC is one of the most widely used production quality compilers available, and its open source nature means that contributions made as the result of this research can have a positive impact for a huge number of users, with GCC being the standard shipped compiler collection for most GNU/Linux systems. Additionally, practical implementations of automatic parallelizing optimisations can serve as references to the development of other compilers such as LLVM.

I feel fortunate in having the opportunity to participate in furthering this field at a time in which software engineering is undergoing such a major tidal shift, perhaps the largest single change in the programming zeitgeist since the

transition from structured to object orientated software abstractions. I was delighted to discover this CDT in a subject that I am so passionate about, and in discovering the supervisors that have research interests so closely aligned with my own. If offered a position as a research student, I would apply the same dedication and conscientious work ethic that has led me to excel in my previous academic and industrial experiences. I would endeavour to make a positive contribution to this excellent University and the field of research.

Bibliography

[1] S. Akhter and J. Roberts. *Multi-core Programming: Increasing Performance Through Software Multi-threading*. Books by engineers, for engineers. Intel Press, 2006. ISBN: 9780976483243.

[2] S. Herb. "The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software". In: *Dr. Dobb's Journal* 30.3 (2005). URL: http://www.gotw.ca/publications/concurrency-ddj.htm.

[3] W. David et al. *Core Count vs Cache Size for Manycore Architectures in the Cloud*. Tech. rep. Cambridge, MA 02139: Massachusetts Institute of Technology, 2010. URL: http://people.csail.mit.edu/beckmann/publications/tech_reports/grain_size_tr_feb_2010.pdf.

[4] *Automatic Parallelization with Intel® Compilers*. Intel Corporation. URL: http://software.intel.com/en-us/articles/automatic-parallelization-with-intel-compilers.

[5] GCC Wiki. *http://gcc.gnu.org/wiki/Graphite/Parallelization*. [Online; accessed 8-April-2014]. 2009. URL: http://gcc.gnu.org/wiki/Graphite/Parallelization.

[6] Gene M Amdahl. "Validity of the single processor approach to achieving large scale computing capabilities". In: *Proceedings of the April 18-20, 1967, spring joint computer conference*. ACM. 1967, pp. 483–485.

[7] R. Ladelsky. *Automatic Parallelization in GCC*. Tech. rep. IBM Haifa Research Lab, 2007.

[8] D. Novillo. "OpenMP and automatic parallelization in GCC". In: *GCC developers summit*. Citeseer. 2006.

[9] Peter MW Knijnenburg, Toru Kisuki, and Michael FP O'Boyle. "Iterative compilation". In: *Embedded processor design challenges*. Springer. 2002, pp. 171–187.

[10] H. Leather, E. Bonilla, and M. O'Boyle. "Automatic Feature Generation for Machine Learning Based Optimizing Compilation". In: *Code Generation and Optimization, 2009. CGO 2009. International Symposium on*. IEEE Computer Society, 2009, pp. 81–91. ISBN: 978-0-7695-3576-0. DOI: 10.1109/CGO.2009.21.

[11] G. Fursin et al. "Milepost GCC: Machine Learning Enabled Self-tuning Compiler". In: *International journal of parallel programming* 39.3 (2011), pp. 296–327. ISSN: 0885-7458. DOI: 10.1007/s10766-010-0161-2.