Compiler Generated Annotations for Run-Time Optimization

Ian Pratt, Tim Harris, Keir Fraser

University of Cambridge Computer Laboratory,
Pembroke Street, Cambridge CB2 3QG, U.K.
Ian.Pratt@cl.cam.ac.uk

1 Introduction

Modern processors rely heavily on 'common-case' assumptions to achieve high instruction throughput. During the course of execution they build up soft state in caches and other prediction units which is used to guide subsequent execution of the instruction stream. Most of the time these caches and predictors work well, yielding high 'hit' rates that result in the processor achieving good functional unit utilization with useful work.

However, the relative cost of a 'miss' is growing. The gap between main-memory latency and CPU performance is widening, and the depth of speculative execution is increasing. The cost of a misprediction can run into hundreds of wasted execution slots, and this trend looks set to increase. Furthermore, it seems unlikely that significant progress will be made improving the hit rate of the various caches and predictors. Due to their hardware implementation and the need for rapid access, the scope for implementing complex algorithms is rather limited.

Improvements in compiler technology may help this situation to some extent, particularly if profiling feedback is employed [1]. Inserting data prefetch instructions can avoid certain data hazards, and predicated execution can be used to replace some hard-to-predict branches. However, such approaches are limited by the dynamic nature of the causes of many stalls. The execution behaviour of code often changes over time, as the program moves between different execution phases. It is also often highly dependent on the program's input data set. These factors make it difficult to provide accurate profiling information to drive feedback-directed optimizations [2]. Even if such information can be provided, the compiler has the difficult task of trying to generate multiple versions of each section of code, and logic to switch between them depending on actual execution behaviour. It seems unlikely that such an approach would prove widely productive.

These problems have led to research into *dynamic optimization* [3], where a run-time system monitors program execution and uses binary re-writing to optimize the application as it runs. Although the technology is relatively immature, several groups have published performance figures that compare favourably with compiler-based feedback systems on selected benchmark applications.

The main focus of current dynamic optimizers is the identification of commonly-executed instruction sequences, and the subsequent optimization of those 'hot' traces. The existing binary is then 'patched' so that program flow will pass to the optimized trace. Traces which are created in this way offer a number of opportunities for optimization. The trace will have been 'straightened', meaning that unconditional control flow instructions can be eliminated, and the predicted path of conditional branches will be to fall through. This can result in improved access density in the instruction cache, assuming the trace really is executed frequently without early exit back to the original code.

Since such traces are typically several hundred instructions long, there is also significant opportunity for peephole optimization and instruction scheduling. More recent dynamic optimizers such as Wiggins/Redstone [4] are able to perform rudimentary *value profiling*, and hence specialise traces for certain *glacial values* – registers that typically contain the same value.

Although these instruction-trace optimizations can sometimes offer significant benefits, current dynamic optimizers are severely restricted in the kinds of transformation it is safe for them to perform. They must ensure that the optimized trace produces exactly the same externally visible effects as the original, because the run-time system has no idea which register and memory values are actually important. For example, care must be taken to ensure that exceptions are exposed in an identical manner. It is also necessary to be careful about the ordering of memory references (even loads) since it has no knowledge of address aliasing, or even which memory locations are visible to hardware devices or other CPUs.

Improved optimization can be obtained by making some assumptions about conventions that code is likely to follow in practice. For example, most programs are written in a high-level language that obeys a calling convention over and above that mandated by the CPU instruction set. Thus it is frequently possible to eliminate register spills when the trace crosses procedure boundaries, as the meaning of those stack manipulations can be inferred by the optimizer. It may also be possible for the optimizer to detect certain other compiler-generated idioms and exploit this knowledge.

Relaxing the optimizer's restrictions may lead to better performance, but can lead to problems with correctness. Assembly code, and programs generated by aggressive compilers that exploit inter-procedural analysis, may fail. Furthermore, there is a real danger that such optimization will expose previously hidden bugs in programs. As a result, it seems unlikely that an effective dynamic optimizer operating solely at the machine-language level would ever be recommended by third party software vendors.

The fundamental problem with the existing approach is that, at the machine-language level, the optimizer doesn't know enough about the high-level operation of the program and the assumptions made by the compiler and language run-time system. It may be able to recognise a small set of idioms and make guesses based on these, but this approach is clearly limited.

2 Proposed Research

It is our belief that an effective optimization system must be tied more closely to the compiler used to generate the application being profiled. Our approach is to allow the compiler to explicitly annotate the programs it produces and thus pass higher-level information and the results of its in depth analyses to the run-time optimizer.

Annotations may include items such as type information (including aliasing and volatility), function and basic block boundaries, along with loop invariants and access stride patterns. These annotations, and the framework for expressing them, will have much in common with those developed for typed assembly language [5] and proof-carrying code [6]. We believe these annotations can supply the runtime with a wealth of information that can be cheaply applied to yield significant performance gains.

In particular, whereas current optimizers are severely limited as regards re-ordering of data references, we believe that appropriate annotations will give scope for useful yet provably safe re-scheduling. It may even be possible to change the layout of data structures in memory to optimize cache performance. Aggressive optimization of a program's data reference stream will become increasingly important as the CPU-memory performance gap widens.

In continuation of our current work on dynamic optimization, we believe that the runtime optimization should be performed concurrently with program execution, in anticipation of Simultaneously Multi-Threaded (SMT) processor architectures. Concurrent hardware threads can be used by the runtime system to optimize running programs.

Annotations may serve a further purpose on SMT architectures to enable efficient exploitation of loop- and thread-level parallelism. For such architectures, the compiler is not in a position to know how many hardware threads it should schedule work for. The "correct" number will depend on factors such as hardware configuration, operating system workload, and inter-thread cache interactions. We intend to address this by enabling the compiler and/or programmer to express as much parallelism as possible, and allow the runtime to select how much to actually exploit based on feedback from the processor and operating system. Fork/join primitives such as those developed for Cilk [7] should enable this to be achieved with low overhead.

A further area we wish to investigate is the interface that processor hardware presents to the software runtime. This consists of two aspects: support for concurrent modification of running programs, and the interface through which the hardware provides performance feedback. It is possible that low-cost hardware modifications might yield substantial cycle savings to the runtime and provide information of better utility.

References

[1] T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of (MICRO-96)*, December 1998.

[2] M. D. Smith. Overcoming the challenges to feedback-directed optimization. *ACM SIGPLAN Notices*, 35(7):1–11, July 2000.

[3] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 1–12, June 2000.

[4] D. Deaver, R. Gorton, and N. Rubin. Wiggins/redstone: An on-line program specializer. In *Proceddings of Hot Chips 11*, August 1999.

[5] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999.

[6] G. Necula. Proof-carrying code. In *Proceedings of the Twenty Fourth ACM Symposium on Principles of Programming Languages*, January 1997.

[7] M. Frigo, C. Leiserson, and K. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 212–223, May 1998.