Aggressive Memory-Aware Compilation

Peter Grun  Nikil Dutt  Alex Nicolau
pgrun@ece.ucr.edu  dutt@ece.ucr.edu  nicolau@ece.ucr.edu

Center for Embedded Computer Systems
University of California, Irvine, CA 92697-3425, USA
http://www.ece.ucr.edu/aces

Abstract

Memory delays represent a major bottleneck in embedded systems performance. Newer memory modules exhibiting efficient access modes (e.g., page-, burst-mode) partly alleviate this bottleneck. However, such features cannot be efficiently exploited in processor-based embedded systems without memory-aware compiler support. We describe a memory-aware compiler approach that exploits such efficient memory access modes by extracting accurate timing information, allowing the compiler’s scheduler to perform global code reordering to better hide the latency of memory operations. Moreover, we present a compiler technique which in the presence of caches actively manages cache misses, and performs global miss traffic optimizations, to better hide the latency of the memory operations. Our memory-aware compiler scheduled several benchmarks on the TI C6201 processor architecture interfaced with a 2-bank synchronous DRAM and generated average improvements of 24% in the presence of efficient access modes, and 61.6% improvement in the presence of caches, over the best possible schedule using a traditional (memory-transparent) optimizing compiler, demonstrating the utility of our memory-aware compilation approach.

1 Introduction

The memory subsystem is one of the key performance and power bottlenecks in emerging architectures: as the gap widens between processors and memories, long memory latencies hinder processor performance, while simultaneously dissipating more power. Advances in memory technology and memory architectures partially alleviate this problem, for instance through new generations of memories such as SDRAM, and Rambus that exhibit efficient access modes (e.g., page-, burst-, and pipelined-access mode). However, these newer memory families still have to be accessed through some form of caching in order to deliver sustained performance. Previous work in optimizing compilers and cache architectures have focused on improving cache performance through exploitation of program locality, program scheduling to hide latencies of cache misses, etc. However, such techniques have traditionally assumed a fairly generic model.

*This work was partially supported by grants from NSF (MIP-9708067), DARPA (F33615-00-C-1632) and a Motorola fellowship.

© Springer-Verlag Berlin Heidelberg 2001
of the background memories being accessed. While these newer memory families are

designed with special, performance-enhancing access modes, even the most aggressive
traditional optimizing compilers are unable to fully exploit such features. Our memory-
aware compilation approach explicitly models and captures detailed timing characteristics
of newer memory families, and exploits this timing information to further improve
processor performance and provide opportunities for power management.

In our approach, we capture the memory access protocols for each memory com-
ponent through a detailed and accurate timing model for the different memory access
modes of a memory component. Using this timing information, our compiler techniques
are able to better match the characteristics of the memory sub-system with the specific
processor architecture, leading to significant improvements in performance. Traditionally,
these access modes were transparent to the processor, and were exploited implicitly
by the memory controller (e.g., whenever a memory access referenced an element al-
ready in the DRAM's row buffer, it avoided the row-decode step, fetching it directly
from the row buffer). However, since the memory controller only has access to local
information, it is unable to perform more global optimizations (such as global code
reordering to better exploit special memory access modes). Our approach provides the
compiler with a more accurate timing model for the specific memory access modes,
and thus allows our compiler to perform global optimizations that aggressively hide
the latency of the memory operations. Moreover, due to the ubiquity of caches in today's
architectures, optimizing the memory access in the presence of caches is crucial. In
the presence of caches, the accurate timing information allows our compiler to explicitly
manage cache miss traffic, generating better performance through the hiding of cache
miss latencies.

Our memory-aware compilation approach exploits detailed memory timing informa-
tion, providing an opportunity to perform global compiler optimizations. First, we ex-
tract accurate memory timing from an architectural description of the processor/memory
system in the EXPRESSION Architectural Description Language (ADL). Then, we use
this detailed memory timing information to efficiently exploit the features of the mem-
ory modules, such as page-mode and burst-mode accesses, pipelining and parallelism.
Additionally, in cache-based architectures we further improve performance through ex-

plicit management of cache miss traffic.

The key idea in our approach is the notion of combining detailed timing of the mem-
ory modules (e.g., efficient memory access modes) with the processor pipeline timings
to generate accurate operation timings. We then use these exact operation timings to
better schedule the application, and hide the latency of the memory operations. Process-
sors traditionally rely on a memory controller to synchronize and utilize specific access
modes of memory modules (e.g., freeze the pipeline when a long delay from a memory
read is encountered). However, the memory controller only has a local view of the (al-
ready scheduled) code being executed. In the absence of an accurate timing model, the
best the compiler can do is to schedule optimistically, assuming the fastest access time
(e.g., page mode, or a hit in the presence of a cache), and rely on the memory controller to account for longer delays, often resulting in performance penalty. This optimistic approach can be significantly improved by integrating an accurate timing model into the compiler. In our approach, we provide a detailed memory timing model to the compiler so that it can better utilize efficient access modes through global code analysis and optimizations, and help the memory subsystem produce even better performance. We use these accurate operation timings in our retargetable compiler to better hide the latency of the memory operations, and obtain further performance improvements.

Moreover, in the absence of dynamic data hazard detection (e.g., in VLIW processors), these operation timings are required to ensure correct behavior: the compiler uses them to insert NOPs in the schedule to avoid data hazards. In the absence of a detailed timing model, the compiler is forced to use a pessimistic schedule, thus degrading overall performance.

2 Related Work

There has been related work in 2 domains: high-level synthesis and mainstream compilers and architectures. In high-level synthesis, Panda et al. [7] present pre-synthesis optimizations to use the page-mode DRAM access. [6] extend this work to Synchronous and RAMBUS DRAMs, using burst-mode accesses, and exploiting memory bank interleaving.

Recent work on interface synthesis [1], [2] present techniques to formally derive node clusters from interface timing diagrams. These techniques can be applied to provide an abstraction of the memory module timings required by our approach.

In [3] and [4] we presented preliminary results on generating and using accurate timing information in the compiler, for page- and burst-mode DRAM accesses, as well as in the presence of caches, hiding the miss latencies by improving the overlap between cache misses and hits to a different cache line.

In the compilers/architectures domain, recent work by Rixner et al. [8] presents a memory controller approach to dynamically reorder the memory accesses, and improve the utilization of the DRAM access modes. The dynamic reordering applies only to a window of pending memory accesses. By performing static compiler optimizations, it is possible to further improve the memory access schedule by globally reordering them. We complement this work by making the compiler aware of the memory access modes and timings. Moreover, we apply a similar compiler approach in the presence of caches.

3 Overview of Experiments

Our experiments demonstrate the performance gains obtained by using accurate timing in the compiler for the Texas Instruments TIC6201 VLIW DSP [9] processor interfaced with the IBM0316409C [5] Synchronous DRAM. We first optimize a set of
benmarks to better utilize the efficient memory access modes (e.g., through memory mapping, code reordering or loop unrolling), and then we use the accurate timing model to further improve the performance by hiding the latencies of the memory operations. To separate out the benefit of the better timing model from the gain due to the access mode optimizations and the access modes themselves, we present the set of results which show the performance gains obtained by scheduling with accurate timing in the presence of a code already optimized for memory accesses, and compare them to the performance of the same memory-access-optimized code using less accurate timing, scheduled optimistically, assuming the shortest access time available (page-mode access), and relying on the memory controller to account for longer delays. This optimistic scheduling is the best alternative available to the compiler, short of an accurate timing model. The performance gains from exploiting detailed memory timing vary from 6% to 47.9%, and an average of 23.9% over a schedule that exploits the efficient access modes without detailed timing. We also compare the above approaches to the baseline performance of the system in the absence of efficient memory access modes.

Our second set of experiments demonstrate the performance gains obtained by aggressively optimizing the memory miss traffic on a set of multimedia and DSP benchmarks. We perform the optimization in two phases: first we isolate the cache misses and attach accurate hit and miss timing to the memory accesses, to allow the scheduler to better target the memory subsystem architecture, obtaining between 15.2% and 52.8% performance improvement over the traditional compiler. We then further optimize the cache miss traffic, by loop shifting to reduce the intra-iteration dependence chains due to accesses to the same cache line, and allow more overlap between memory accesses, resulting in a further 21.3% average performance improvement.

Currently, our work applies to wide issue statically scheduled VLIW Processors, and preliminary results have been presented at DAC-2000 [3] and ICCAD-2000 [4]. We believe that our techniques are also applicable to dynamically scheduled processors. Our on-going work evaluates the improvements of our approach for out-of-order issue superscalar processors, and also addresses the tradeoff between increase in code size (due to loop unrolling) versus performance improvement.

References


