DOWNLOAD PDF CODE OPTIMIZATION TECHNIQUES FOR EMBEDDED PROCESSORS METHODS, ALGORITHMS, AND TOOLS

Chapter 1 : Program optimization - Wikipedia

*Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools [Rainer Leupers] on blog.quintoapp.com *FREE* shipping on qualifying offers. The building blocks of today's and future embedded systems are complex intellectual property components, or cores.*

SystemCoDesigner offers a fast design space exploration and rapid prototyping of behavioral SystemC models. Together with Forte Design Systems, a fully automated approach was developed by integrating behavior Together with Forte Design Systems, a fully automated approach was developed by integrating behavioral synthesis into the design flow. Starting from a behavioral SystemC model, hardware accelerators can be generated automatically using Forte Cynthesizer and can be added to the design space. The resulting design space is explored automatically by optimizing several objectives simultaneously using state of the art multi-objective optimization algorithms. Abstractâ€”Digital signal processing DSP applications involve processing long streams of input data. It is important to take into account this form of processing when implementing embedded software for DSP systems. Task-level vectorization, or block processing, is a useful dataflow graph transformat Task-level vectorization, or block processing, is a useful dataflow graph transformation that can significantly improve execution performance by allowing subsequences of data items to be processed through individual task invocations. In this way, several benefits can be obtained, including reduced context switch overhead, increased memory locality, improved utilization of processor pipelines, and use of more efficient DSPoriented addressing modes. On the other hand, block processing generally results in increased memory requirements since it effectively increases the sizes of the input and output values associated with processing tasks. In this paper, we investigate the memory-performance tradeoff associated with block processing. We develop novel block processing algorithms that take carefully take into account memory constraints to achieve efficient block processing configurations within given memory space limitations. Our experimental results indicate that these methods derive optimal memory-constrained block processing solutions most of the time. We demonstrate the advantages of our block processing techniques on practical kernel functions and applications in the DSP domain. Show Context Citation Context Task-level vectorization or block processing is one general method for improving DSP software performance in a variety of ways. In this context, block processing refers to the ability of a task to This paper stresses the importance of designing efficient embedded software and it provides a global view of some of the techniques that have been developed to meet this goal. These techniques include high-level transformations, compiler optimizations reducing the energy consumption of embedded prog These techniques include high-level transformations, compiler optimizations reducing the energy consumption of embedded programs and optimizations exploiting architectural features of embedded processors. Such optimizations lead to significant reductions of the execution time, the required energy and the memory size of embedded applications. Despite this, they can hardly be found in any available compiler. Therefore, two 32 bit data types, four 16 bit data types o Acknowledgments by Sai Pinnepalli , " This dissertation could not have been completed without significant help and input from two people. First, I would like to thank Dr. Ramanujam , who guided me with patience and accommodated my schedule to help me with this work. Second, I would like to thank Dr. Doris Carver, who steadfastly Doris Carver, who steadfastly directed me towards this goal. I would also like to thank Dr. Jinpyo Hong, and Dr. Thomas Shaw for serving on my committee. I would be remiss if I did not mention the amount of time Dr. Hong spent on weekends discussing my work. Ram has the ability to discuss your ideas as if every one of them merits discussion. I am grateful for these discussions, some of which are chapters in this dissertation. Working full time while trying to pursue this degree required cooperation from my employers. Carol Wesson actively supported my pursuit. My appreciation for their support is heartfelt. The objective of this work is to create a framework for the optimization of embedded software. We present algorithms and a tool flow to reduce the computational effort of programs, using value profiling and partial evaluation. Such a

reduction translates into both energy savings and average-case per Such a reduction translates into both energy savings and average-case performance improvement, while preserving a tolerable increase of worst case performance and code size. Our tool reduces the computational effort by specializing frequently executed procedures for the most common values of their parameters. The most effective specializations are automatically searched and identified, and the code is transformed through partial evaluation. Also, their automatic search engine greatly reduces code optimization time with respect to exhaustive search. In the last three decades the world of computers and especially that of microprocessors has been advanced at exponential rates in both productivity and performance. The integrated circuit industry has followed a steady path of constantly shrinking devices geometries and increased functionality that The integrated circuit industry has followed a steady path of constantly shrinking devices geometries and increased functionality that larger chips provide. The technology that enabled this exponential growth is a combination of advancements in process technology, microarchitecture, architecture and design and development tools. This paper overviews some of the microarchitectural techniques that are typical for contemporary high-performance microprocessors. The techniques are classified into those that increase the concurrency in instruction processing, while maintaining the appearance of sequential processing pipelining, super-scalar execution, out-of-order execution, etc. In addition, the paper also discusses microarchitectural techniques likely to be used in the near future such as microarchitectures with multiple sequencers and thread-level speculation, and microarchitectural techniques intended for minimization of power consumption. Typical applications of ESs include medical electronics pacemakers , personal communication devices wireless phones , automobiles antilock braking systems , aviation fly-by-wire flight control s

DOWNLOAD PDF CODE OPTIMIZATION TECHNIQUES FOR EMBEDDED PROCESSORS METHODS, ALGORITHMS, AND TOOLS

Chapter 2 : CiteSeerX â€" Citation Query Code Optimization Techniques for Embedded

Code Optimization Techniques for Embedded Processors: Methods, Algorithms, and Tools / Edition 1 The building blocks of today's embedded systems-on-a-chip are complex IP components and programmable processor cores.

This paper presents techniques to tightly integrate worstcase execution time WCET information into a compiler framework. Currently, a tight integration of WCET information into the compilation process is strongly desired, but only some ad-hoc approaches have been reported currently. Previous publications mainly used self-written WCET estimators with very limited functionality and preciseness during compilation. A very tight integration of a high quality industry-relevant WCET analyzer into a compiler was not yet achieved up to now. This work is the first to present techniques capable of achieving such a tight coupling between a compiler and the WCET analyzer aiT. Additionally, the results produced by the WCET analyzer are automatically collected and re-imported into the compiler infrastructure. The work described in this paper is smoothly integrated into a C compiler environment for the Infineon TriCore processor. It opens up new possibilities for the design of WCET-aware optimizations in the future. The concepts for extending the compiler infrastructure are kept very general so that they are not limited to WCET information. Rather, it is possible to use our structures also for multi-objective optimization of e. Show Context Citation Context Modern compilers include a vast variety of optimizations. However, they aim at minimizing e. The effect of optimizations on WCET is almost fully unknown. Currently, the executable produced by the compiler is manually fed into a WCET analyzer computing timing infor Abstract â€" This paper describes a set of novel highlevel control flow transformations for performance improvement of typical address-dominated multimedia applications. We show that these transformations applied at the source code level can have a very large impact on execution time at the cost of li We show that these transformations applied at the source code level can have a very large impact on execution time at the cost of limited overhead in code size for a broad range of instruction set processor families i. For a profound evaluation, all transformations are applied to the C-codes of two real-life applications selected from the video and image processing domains. A detailed analysis of the effect of the transformations is done by compiling and executing the transformed programs on seven different programmable processors. The measured runtimes indicate quite significant improvements in all processor families when comparing the performance of the transformed codes to their initial version even when these are compiled using their native optimizing compilers with their most aggressive optimization features enabled. The average gains in execution time range from This way, optimizations based on th

DOWNLOAD PDF CODE OPTIMIZATION TECHNIQUES FOR EMBEDDED PROCESSORS METHODS, ALGORITHMS, AND TOOLS

Chapter 3 : Code Optimization Techniques for Embedded Processors : Rainer Leupers :

Code Optimization Techniques for Embedded Processors discusses the state-of-the-art in the area of compilers for embedded processors. It presents a collection of new code optimization techniques, dedicated to DSP and multimedia processors.

General[edit] Although the word "optimization" shares the same root as "optimal", it is rare for the process of optimization to produce a truly optimal system. The optimized system will typically only be optimal in one application or for one audience. One might reduce the amount of time that a program takes to perform some task at the price of making it consume more memory. In an application where memory space is at a premium, one might deliberately choose a slower algorithm in order to use less memory. Often there is no "one size fits all" design which works well in all cases, so engineers make trade-offs to optimize the attributes of greatest interest. Fortunately, it is often the case that the greatest improvements come early in the process. Levels of optimization[edit] Optimization can occur at a number of levels. Typically the higher levels have greater impact, and are harder to change later on in a project, requiring significant changes or a complete rewrite if they need to be changed. Thus optimization can typically proceed via refinement from higher to lower, with initial gains being larger and achieved with less work, and later gains being smaller and requiring more work. However, in some cases overall performance depends on performance of very low-level portions of a program, and small changes at a late stage or early consideration of low-level details can have outsized impact. Typically some consideration is given to efficiency throughout a project â€” though this varies significantly â€” but major optimization is often considered a refinement to be done late, if ever. On longer-running projects there are typically cycles of optimization, where improving one area reveals limitations in another, and these are typically curtailed when performance is acceptable or gains become too small or costly. As performance is part of the specification of a program â€” a program that is unusably slow is not fit for purpose: This is sometimes omitted in the belief that optimization can always be done later, resulting in prototype systems that are far too slow â€” often by an order of magnitude or more â€” and systems that ultimately are failures because they architecturally cannot achieve their performance goals, such as the Intel ; or ones that take years of work to achieve acceptable performance, such as Java , which only achieved acceptable performance with HotSpot The degree to which performance changes between prototype and production system, and how amenable it is to optimization, can be a significant source of uncertainty and risk. The architectural design of a system overwhelmingly affects its performance. For example, a system that is network latency-bound where network latency is the main constraint on overall performance would be optimized to minimize network trips, ideally making a single request or no requests, as in a push protocol rather than multiple roundtrips. Choice of design depends on the goals: Choice of platform and programming language occur at this level, and changing them frequently requires a complete rewrite, though a modular system may allow rewrite of only some component â€” for example, a Python program may rewrite performance-critical sections in C. In a distributed system, choice of architecture client-server , peer-to-peer , etc. Algorithms and data structures Given an overall design, a good choice of efficient algorithms and data structures , and efficient implementation of these algorithms and data structures comes next. After design, the choice of algorithms and data structures affects efficiency more than any other aspect of the program. Generally data structures are more difficult to change than algorithms, as a data structure assumption and its performance assumptions are used throughout the program, though this can be minimized by the use of abstract data types in function definitions, and keeping the concrete data structure definitions restricted to a few places. For algorithms, this primarily consists of ensuring that algorithms are constant O 1 , logarithmic O log n , linear O n , or in some cases log-linear O n log n in the input both in space and time. Algorithms with quadratic complexity O n2 fail to scale, and even linear algorithms cause problems if repeatedly called, and are typically replaced with constant or logarithmic if possible. Beyond asymptotic order of growth, the

constant factors matter: Often a hybrid algorithm will provide the best performance, due to this tradeoff changing with size. A general technique to improve performance is to avoid work. A good example is the use of a fast path for common cases, improving performance by avoiding unnecessary work. For example, using a simple text layout algorithm for Latin text, only switching to a complex layout algorithm for complex scripts, such as Devanagari. Another important technique is caching, particularly memoization , which avoids redundant computations. Because of the importance of caching, there are often many levels of caching in a system, which can cause problems from memory use, and correctness issues from stale caches. Source code level Beyond general algorithms and their implementation on an abstract machine, concrete source code level choices can make a significant difference. For example, on early C compilers, while 1 was slower than for ;; for an unconditional loop, because while 1 evaluated 1 and then had a conditional jump which tested if it was true, while for ;; had an unconditional jump. Some optimizations such as this one can nowadays be performed by optimizing compilers. This depends on the source language, the target machine language, and the compiler, and can be both difficult to understand or predict and changes over time; this is a key place where understanding of compilers and machine code can improve performance. Loop-invariant code motion and return value optimization are examples of optimizations that reduce the need for auxiliary variables and can even result in faster performance by avoiding round-about optimizations. Build level Between the source and compile level, directives and build flags can be used to tune performance options in the source code and compiler respectively, such as using preprocessor defines to disable unneeded software features, optimizing for specific processor models or hardware capabilities, or predicting branching, for instance. Compile level Use of an optimizing compiler tends to ensure that the executable program is optimized at least as much as the compiler can predict. Assembly level At the lowest level, writing code using an assembly language , designed for a particular hardware platform can produce the most efficient and compact code if the programmer takes advantage of the full repertoire of machine instructions. Many operating systems used on embedded systems have been traditionally written in assembler code for this reason. Programs other than very small programs are seldom written from start to finish in assembly due to the time and cost involved. Most are compiled down from a high level language to assembly and hand optimized from there. When efficiency and size are less important large parts may be written in a high-level language. With more modern optimizing compilers and the greater complexity of recent CPUs , it is harder to write more efficient code than what the compiler generates, and few projects need this "ultimate" optimization step. Much code written today is intended to run on as many machines as possible. Additionally, assembly code tuned for a particular processor without using such instructions might still be suboptimal on a different processor, expecting a different tuning of the code. Typically today rather than writing in assembly language, programmers will use a disassembler to analyze the output of a compiler and change the high-level source code so that it can be compiled more efficiently, or understand why it is inefficient. Run time Just-in-time compilers can produce customized machine code based on run-time data, at the cost of compilation overhead. This technique dates to the earliest regular expression engines, and has become widespread with Java HotSpot and V8 for JavaScript. In some cases adaptive optimization may be able to perform run time optimization exceeding the capability of static compilers by dynamically adjusting parameters according to the actual input or other factors. Profile-guided optimization is an ahead-of-time AOT compilation optimization technique based on runtime profiles, and is similar to a static "average case" analog of the dynamic technique of adaptive optimization. Self-modifying code can alter itself in response to run time conditions in order to optimize code; this was more common in assembly language programs. Some CPU designs can perform some optimizations at runtime. Some examples include Out-of-order execution , Speculative execution , Instruction pipelines , and Branch predictors. Compilers can help the program take advantage of these CPU features, for example through instruction scheduling. Platform dependent and independent optimizations[edit] Code optimization can be also broadly categorized as platform -dependent and platform-independent techniques. While the latter ones are effective on most or all platforms, platform-dependent techniques use specific properties of one platform, or rely on parameters

depending on the single platform or even on the single processor. Writing or producing different versions of the same code for different processors might therefore be needed. For instance, in the case of compile-level optimization, platform-independent techniques are generic techniques such as loop unrolling , reduction in function calls, memory efficient routines, reduction in conditions, etc. A great example of platform-independent optimization has been shown with inner for loop, where it was observed that a loop with an inner for loop performs more computations per unit time than a loop without it or one with an inner while loop. On the other hand, platform-dependent techniques involve instruction scheduling, instruction-level parallelism , data-level parallelism, cache optimization techniques i. Strength reduction[edit] Computational tasks can be performed in several different ways with varying efficiency. A more efficient version with equivalent functionality is known as a strength reduction. For example, consider the following C code snippet whose intention is to obtain the sum of all integers from 1 to N: See algorithmic efficiency for a discussion of some of these techniques. However, a significant improvement in performance can often be achieved by removing extraneous functionality. Optimization is not always an obvious or intuitive process. In the example above, the "optimized" version might actually be slower than the original version if N were sufficiently small and the particular hardware happens to be much faster at performing addition and looping operations than multiplication and division. Trade-offs[edit] In some cases, however, optimization relies on using more elaborate algorithms, making use of "special cases" and special "tricks" and performing complex trade-offs. A "fully optimized" program might be more difficult to comprehend and hence may contain more faults than unoptimized versions. Beyond eliminating obvious antipatterns, some code level optimizations decrease maintainability. Optimization will generally focus on improving just one or two aspects of performance: For example, increasing the size of cache improves runtime performance, but also increases the memory consumption. Other common trade-offs include code clarity and conciseness. There are instances where the programmer performing the optimization must decide to make the software better for some operations but at the cost of making other operations less efficient. Such changes are sometimes jokingly referred to as pessimizations. Bottlenecks[edit] Optimization may include finding a bottleneck in a system â€” a component that is the limiting factor on performance. More complex algorithms and data structures perform well with many items, while simple algorithms are more suitable for small amounts of data â€” the setup, initialization time, and constant factors of the more complex algorithm can outweigh the benefit, and thus a hybrid algorithm or adaptive algorithm may be faster than any single algorithm. A performance profiler can be used to narrow down decisions about which functionality fits which conditions. For example, a filtering program will commonly read each line and filter and output that line immediately. This only uses enough memory for one line, but performance is typically poor, due to the latency of each disk read. Performance can be greatly improved by reading the entire file then writing the filtered result, though this uses much more memory. Caching the result is similarly effective, though also requiring larger memory use. When to optimize[edit] Optimization can reduce readability and add code that is used only to improve the performance. This may complicate programs or systems, making them harder to maintain and debug. As a result, optimization or performance tuning is often performed at the end of the development stage. Donald Knuth made the following two statements on optimization: This can result in a design that is not as clean as it could have been or code that is incorrect, because the code is complicated by the optimization and the programmer is distracted by optimizing. A simple and elegant design is often easier to optimize at this stage, and profiling may reveal unexpected performance problems that would not have been addressed by premature optimization. In practice, it is often necessary to keep performance goals in mind when first designing software, but the programmer balances the goals of design and optimization. Modern compilers and operating systems are so efficient that the intended performance increases often fail to materialize. As an example, caching data at the application level that is again cached at the operating system level does not yield improvements in execution. Even so, it is a rare case when the programmer will remove failed optimizations from production code. It is also true that advances in hardware will more often than not obviate any potential improvements, yet the obscuring code

will persist into the future long after its purpose has been negated. Macros[edit] Optimization during code development using macros takes on different forms in different languages. Nowadays, inline functions can be used as a type safe alternative in many cases. In both cases, the inlined function body can then undergo further compile-time optimizations by the compiler, including constant folding , which may move some computations to compile time. Since in many cases interpretation is used, that is one way to ensure that such computations are only performed at parse-time, and sometimes the only way.

Chapter 4 : NASA - Engineering Tools

Code Optimization Techniques for Embedded Processors Methods, Algorithms, and Tools. Authors: Leupers, Rainer Buy this book eBook ,