Optimization of memory system in Real-Time Embedded Systems

A R MAHAJAN
Department of Computer Science
Nagpur University,
Nagpur,
INDIA

M S ALI
Department of Computer Science
Amravati University,
Amravati
INDIA

Abstract : - Code space is a critical issue in designing of software for real-time embedded systems. The memory system often determines a great deal about the behavior of an embedded system: performance, power, and manufacturing cost. A great many software techniques have been developed to optimize software to improve these characteristics. Since much of the code for embedded systems is compiled once and then burned into ROM, the software designer will often tolerate much longer compile times in the hope of reducing the size of the compiled code. This paper surveys techniques for optimizing memory behavior of real-time embedded software .

Key-words : - embedded systems, real-time, memory, compiler , optimization , cache

1 Introduction

Embedded Real Time systems have to correctly implement the required functionality , as well as , they have to fulfill a wide range of constraints : development cost, unit cost, reliability, security, size, performance, power consumption etc. Critical to the correct functioning of such systems are their timing constraint[31].

A real time system is one that fails if it's performance criteria are not met. Real time systems have been classified as hard and soft. There is a third category , firm real time systems, whose definition falls between those of hard and soft real time system.

A hard real-time system is one that must meet its performance objectives every time and all the time. As soon as one of these systems does not meet one of its performance criteria, it fails. An example of a hard real-time system is a fly-by-wire flight control system, where if the system does not respond to a pilot's commands within microseconds, then the system fails with potentially catastrophic circumstances.

A soft real-time system is one that must meet its performance objectives on average only. This means that if every now and then a performance deadline is missed, the system does not fail. If, however, the system repeatedly misses its performance deadlines, then it fails. An example of a soft real-time system is a streaming media player, where if the system does not meet its performance objectives in a single instance, then the buffered information ensures that there is no loss of information. Should this loss continue over time, however, the quality of the connection becomes reduced and may eventually be lost[32,30].

2. Distributed Real-Time Embedded Systems

Currently, distributed real-time systems are implemented using architectures where each node is dedicated to the implementation of a single function or class of functions. The complete system can be, in general, composed of several networks interconnected with each other (see Figure 1). Each network has its own communication protocol, and internetwork communication is via a *gateway*, which is a node connected to both networks. The architecture can contain several such networks having different types of topologies.

A network is composed of several different types of hardware components, called *nodes*. Typically, every node, also called an *electronic control unit* (ECU), has a communication controller, CPU, RAM, ROM, and an I/O interface to sensors and actuators. Nodes can also have ASICs in order to accelerate parts of their functionality. The microcontrollers used in a node and the type of network protocol employed are influenced by the

nature of the functionality and the imposed real time, fault-tolerance, and power constraints[1].

For processor-based embedded systems, however, the use of compilers is less common. Instead, designers still use assembly language to program many embedded applications. This is because of the high-efficiency requirements of the embedded systems. Processor-based embedded systems frequently employ domain-specific or application-specific instruction set processors (ASIPs), which meet design constraints such as performance, cost, and power consumption more efficiently than general-purpose processors. Building the required software development tool infrastructure for ASIPs, however, is expensive and time-consuming.

The processor architecture and the embedded software executed on the processor must be efficient. The cause of many compilers' poor code quality is the highly specialized architecture of ASIPs, whose instruction sets are incompatible with high-level languages and traditional compiler technology. To combat poor code quality, compiler designers need domain-specific code optimization techniques that go beyond classical compiler technology, which mainly supports machine-independent optimization and code generation for clean architectures (those with homogeneous register files) such as reduced-instruction-set computers (RISCs). Unlike compilers for desktop computers, compilers for ASIPs need not be very fast. Most Embedded software developers agree that a slow compiler is acceptable, provided that it generates efficient code. A compiler can exploit an increased amount of compilation time by using more-effective (and more time-consuming) optimization techniques like genetic algorithms, simulated annealing, integer linear programming, and branch-and-bound search.

The multiple phases of compilers must execute in some order, and each phase can impose unnecessary restrictions on subsequent phases.

2.1 Speed and size

In fact, the speed versus size tradeoff is a critical part of the application build process. Much of the work by an embedded systems programmer is taken up tuning compiler optimization switches for the best mix of performance and compact code size. Using a performance profiler and knowledge of the application, the programmer chooses the time critical parts to optimize for speed, and chooses the rest to optimize for size. If we believe the 90-10 rule, that 90% of the time is spent in 10% of the code, then optimizing that 10% of the code for speed and the rest for size should give the best of both worlds. This doesn't reflect reality, however. There are other reasons to perhaps favor a slightly slower, smaller program, or to favor a slightly larger, faster program.

In Real-Time Embedded Systems both speed and size are significant. Typically a compiler places primary emphasis on the execution speed of the compiled code and much less emphasis on the size of the compiled code. Optimization for execution speed , in some cases, conflict with optimization for code size [16].

2.2 Real-Time requirements

One of the key issues in real-time systems is the schedulability analysis to determine whether the timing constraints will be satisfied at run-time or not. Regardless of which analysis and scheduling techniques are used, it is essential that the real-time designer be able to determine the worst-case execution time of all the activities involved in the application. Moreover, real-time applications run for large periods of time. During its operation, memory can be allocated and deallocated many times which aggravates the memory fragmentation problem. Considering these aspects, the requirements of real-time applications regarding dynamic memory can be stated as follows:

- Bounded response time. The WCRT has to be known a priority and, if possible, be independent of application data. This is the main requirement the must be meet.
- Fast response time. Although having a bounded response time is a must, the response time has to be fast to be usable. A bounded DSA algorithm which is 10 times slower than a conventional one, is not practical.
- Memory requests have to be always satisfied. In non-real time systems applications can receive a null pointer or are just killed by the OS when the system runs out of memory. It is obvious that is not possible to always grant all the memory requested.

Although there is large range of real-time systems with different memory constraints and hardware support, the study presented in this work in progress is focused on embedded systems where memory is a scarce resource and there is no MMU support.

In distributed environment , efforts are made to shift the computation to the disk subsystem eg. Active disks[26,3], intelligent disks[4], smart disks[14], by performing some filtering type of computation on the storage device itself. [15] demonstrates how several database operations can be performed by the embedded processor attached to the storage device.The memory hierarchy has served as a central component in computing platforms since the introduction of Von Neumann Machine. It uses several levels of cache memories, with each level trading off capacity for the access speed, to bridge the widening performance gap between processor and main memory, while processing speeds have nearly doubled every year, memory response times have increased by a much slower rate a year.

3 Embedded Software and Memory

In embedded systems , the cost of the memory hierarchy limits it's ability to play as central a role . This is due to stringent area, power and cost constraints that fundamentally impact design choices, and limit the physical size and complexity of the memory system. Ultimately, application developers and system engineers are charged with the burden of reducing the memory requirements of an application in order to avoid memory bottleneck that degrade processor throughput.

In Real-Time Embedded Systems it can be challenging to fit the needed functionality into the available code space. Economic consideration often dictate the use of a small and cheap processor, while demands of functionality lead to a need for considerable code space. Most compilers ignore the problems of limited code space in Embedded systems. Designers of Embedded systems often have no better alternative than to manually reduce the size of the source code or even the compiled code.

A foundation for the automatic optimization of memory requirements using novel compiler techniques that are designed to increase the synergy between the processor and its memory system is presented in [21]. This thesis offers the possibility of compiler optimization playing a significant role in optimizing the memory design of an ES.

As per [18] , compiler methods are preferable to programmer directives for three reasons : they do not require programmer effort, are portable across different systems and are likely to make better decisions, especially for large , complex programs.

The methods that aim to allocate data to on-chip and off-chip memories mapped to different portions of the address space, are discussed in [19] and [20]. In [17], a complier algorithm for managing scratch-pad based system is presented. The algorithm is able to change the allocation at run time and avoid the overhead of caching. The runtime using this algorithm improves as compared to static allocation. Thus, there is a decrease in energy consumption. As energy consumption is known to be roughly proportional to runtime when the architecture is unchanged.

Usually, embedding a computer in a larger system imposes nonfunctional requirements on the software and hardware, such as hard performance goals (deadlines), power consumption, or code size. It is these constraints that are of primary interest when we shape the memory access patterns of embedded software. In many cases, the memory system is the primary limitation on the performance and power consumption of the embedded software. And, of course, these goals may be mutually incompatible a performance optimization may increase power consumption.

Memory systems often dominate the power consumption of embedded systems. The memory system power consumption is of particular importance in battery-powered embedded systems. [9] showed that off-chip memory access dominates the power consumption of signal processing–oriented embedded systems. In some cases, even excessive cache behavior can cause significant excess power consumption.

Most embedded systems today use a single level of cache (as shown in Fig.1 [33]), but we can expect to see multiple levels of caching as performance needs go up. The main memory system may be contained partially on-chip and partially off-chip. Scratch pad memories (SPMs) have been proposed as one form of high-speed on-chip memory. Off-chip, a variety of technologies may be used, including synchronous dynamic random access memory (SDRAM) or Rambus DRAM (RDRAM). The caches, on-chip main memory, and off-chip main memory can all influence the performance and power consumption of embedded systems. Optimizing software to minimize off-chip references is particularly important for power consumption.

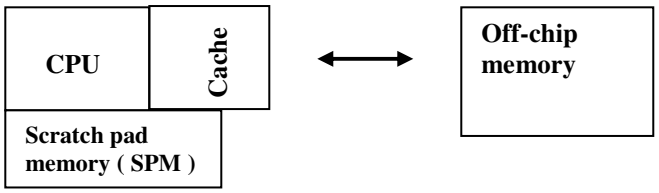

Fig 1 . A memory- centric view of an embedded computing system

Code size is often determined by static properties.We have several advantages when optimizing the software for an embedded system. First, we know the characteristics of the underlying hardware platform and are willing to optimize to that platform. We can even tune the software for such detailed characteristics as the size and organization of the cache. Second, we generally have a more complete set of the software that we can jointly optimize. Because real-time deadlines require global analysis, we have to have all the software that can affect those deadlines in order to be sure that the deadlines are met. As a result, we can optimize the entire software package rather than independently optimizing the pieces. Third, we are generally willing to spend more CPU time on compilation and optimization. Using more CPU time allows us to apply much stronger algorithms to tackle optimization challenges.

Memory optimizations for embedded software comes from two main sources: scientific compilers and hardware/software codesign. The scientific compilation literature has a long, rich history of software optimizations that help relieve memory bottlenecks . Scientific compilers are especially adept at optimizing the data behavior of programs, since array manipulation is at the core of many scientific codes. The main limitation of these algorithms when applied to embedded software is that they use very simple cost models for performance and they do not consider power consumption at all. Joint design of hardware and software is important to achieving the best designs; however, experience shows that when faced with a choice between changing the hardware and changing the software, software optimizations are often easier to do, and are easier to undo if unforeseen side effects occur.

3.1 Data oriented Optimization

Two major tasks in the design of data-oriented memory operations are scheduling and allocation. Loop transformations change the schedule of memory operations by changing the order in which memory accesses to array elements occur. Data layout transformations change the allocation of data by modifying where array elements lie in the cache and in main memory partitions.

Loop transformations are an important tool in data cache Optimization. [13] discussed loop interchange as a means of optimizing parallelism. Techniques for determining the complex loop bounds after loop-oriented program transformations have been proposed by [12]. [11] were the first ones who used transformations such as tiling for improving spatial and temporal reuse (mainly for improving the paging performance in a virtual memory-based environment). [7] and [22] introduced systematic ways of using transformations such as loop permutations, skewing, and tiling for optimizing cache locality. Later, [23] extended unimodular transformations to nonunimodular transformations and demonstrated how complex transformations can be represented within an optimizing compiler. [28], [29], and [24] showed how to use data transformations for locality and integrated them with loop-oriented techniques.

Compiler optimizations that target improving data locality can be divided into two broad groups as discussed in [33] : optimizations that target *reducing* memory latency; and optimizations that target *hiding* memory latency.

3.1.1 Latency-Reducing Transformations

In the first group are the techniques that modify program access pattern, memory layout of variables, or both. Many embedded applications from image and video processing domain focus on loop nests and arrays of signals.

A majority of the proposed techniques target loop-nest-based embedded codes and try to improve locality using loop (iteration space) and/or data layout (memory space) transformations. Loop transformations modify the traversal order of loop iteration points of a given code to obtain a better data locality behavior than the one exhibited by the original code. Among the most popular loop

transformations are loop permutation and loop fusion [19]. Loop permutation changes the order of the loops in a given nest.

For example, consider the following Jacobi iteration code from an image processing application:

```
for (J= 1;J N-1;J++)
for (I=1;I N-1;I++)
B[I][J] = (A[I-1][J]+A[I+1][J]+A[I][J-
1]+A[I][J+1]) * (1/k);
```

Note that, for a fixed value of the loop, the successive iterations of the inner loop in this code access elements from different rows, and this occurs for each reference in the loop body. Since multidimensional arrays are stored in C in the row-major format, this is not a desirable access pattern as far as locality is concerned. An optimizing compiler can transform this code to obtain the following nest:

```
for (I=1;I N-1;I++)
for (J=1;J N-1;J++)
B[I][J] =(A[I-1][J]+A[I+1][J]+A[I][J-
1]+A[I][J+1]) *(1/k);
```

Now, for each reference, the successive iterations of the inner loop access elements from the same row (although different references can access different rows). Therefore, we can expect a much better data locality behavior from this transformed code. Loop permutation belongs to a set of unified class of transformations called unimodular transformations.

Unimodular transformations include loop permutation, loop reversal, and loop skewing, and target a single nest at a time. Loop fusion is, on the other hand, a multinest optimization.

As an example, consider the following code fragment:

```
for (I=0;I N;I++)
k = k + A[I] * B[I];
for (I=0;I N;I++)
c = c * (A[I]+B[I]);
```

Given a small-capacity cache, this fragment can read each element of arrays and into cache twice, once for each loop. We can reduce the number of cache loads by half by fusing these two loops as follows:

```
for (I=0;I N;I++)
{
k = k + A[I] * B[I];
c = c * (A[I]+B[I]);
}
```

Advantage of loop transformation is that each nest in the program can be optimized independently from the others. Loop transformations, however, are restricted by data dependences. In some cases, the best possible transformation from a data locality perspective cannot be applied because it violates an intrinsic data dependence in the code. Also, when a loop transformation is applied, typically the locality behavior of all references in the code are affected. Therefore, it may not be very trivial to find a loop transformation that optimizes all (or, at least, most) of the references in loop body.

Alternative optimization is data space (memory layout) transformations, or data transformations for short [28]. In applying a data transformation, instead of the loop access pattern, we modify the underlying memory layouts of the arrays involved. Implementing a data transformation involves transforming the array subscript expressions and modifying the corresponding array declaration. Consider, for example, the following code fragment:

```
for (I=1;I N-1;I++)
for (J=1;J M-1;J++)
B[J][I] = B[J+1][I-1] +1;
```

Assuming row-major memory layouts as in C, interchanging the order of the two loops shown in this code is not legal (owing to data dependences). However, a data transformation can transform the layout of the array in the code from row-major (the default layout) to column-major. This can be done by modifying the subscript expressions as follows:

```
for (I=1;I N-1;I++)
for (J=1;J M-1;J++)
B'[I][J] = B'[I-1][J+1] +1;
```

Note that the order of the loops is not modified.

3.1.2 Latency-Hiding Transformations

The techniques discussed try to improve data locality, thereby reducing the number of accesses to slower levels in the memory hierarchy. Software prefetching, in contrast, tries to bring data to the fast memory ahead of time (i.e., before the data is actually needed). If this can be chieved, the data can be accessed from cache memory when it is needed. To issue the prefetches enough iterations ahead of their use, a compiler transformation known as software pipelining need to be used. [8] presents a compiler-directed software prefetching algorithm which consists of the following major steps.

- Determine for each array reference the accesses that are likely to be cache misses and therefore need to be prefetched.
- Isolate the predicted cache miss instances through loop splitting [6].

- Apply software pipelining and insert prefetch instructions in the code.

The experimental results using a set of array-intensive applications show that, in most cases, their selective prefetching scheme performs noticeably better than indiscriminate prefetching (that is, issuing a prefetch request for each array reference). These performance benefits (over the indiscriminate prefetching) comes primarily from a reduction in prefetching overhead while still maintaining a comparable savings in memory stall time. Their results also indicate that compiler-directed software prefetching is, in many cases, complementary to classical latency reducing locality optimizations such as iteration space tiling. [5] developed a theory for software-assisted cache replacement. They developed theorems that describe when kill and keep instructions can be inserted into programs such that the modified code will have at least as high a hit rate as before. They also developed a compiler algorithm based on this theory.

4 Conclusion

Efficient analysis and optimization methods are needed and can be developed for the implementation of distributed real-time applications. Real-time embedded system optimization has to tightly integrate software and hardware schemes . Use of more effective optimization techniques for embedded systems compilers will increase the efficiency of these applications and will solve the memory reference problem in embedded systems .

Loop optimizations with memory layout transformations will be very important as embedded applications become more complex and more data intensive.

References:

[1] Paul Pop, Petru Eles, Zebo Peng, and Traian Pop , "Analysis and Optimization of Distributed Real-Time Embedded Systems" , *ACM Transactions on Design Automation of Electronic Systems*, Vol. 11, No. 3, July 2006, Pages 593–625.

[2] C.Kozyrakis, D.Judd, J. Gebis, S. Williams, D. Patterson, and K.Yelick, "Hardware/Compiler Codevelopment for an Embedded Media Processor"

[3] A. Acharya, M. Uysal and J. Saltz, "Active Disks: Programming Model, Algorithms and Evaluation". In Proc. the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, October 1998.

[4] K. Keeton, D. Patterson and J. Hellerstein, "A Case for Intelligent Disks (IDISKs)". In SIGMOD Record, 27(3), 1998.

[5] P. Jain, S. Devadas, D. Engels, and L. Rudolph, "Software-assisted cache replacement mechanisms for embedded systems," in *Proc. Int.Conf. Comput. Aided Design (ICCAD-01)*, 2001, pp. 119–126.

[6] M. Wolfe, *High Performance Compilers for Parallel Computing*, CA: Addison Wesley, 1996.

[7] M. Wolf and M. Lam, "A data locality optimizing algorithm," in *Proc. ACMConf. Program. Lang. Design Implementation*, 1991, pp.30–44.

[8] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLUS V)*, 1992, pp. 62–73.

[9] F. Catthoor, S. Wuytack, E. De Greef , F. Balasa , L. Nachtergaele , and A. Vandecapelle, *Custom Memory Management Methodology: Exploration of Memory Organization for Embedded Multimedia System Design*. Boston, MA: Kluwer, 1998.

[10] M. Kandemir and I. Kadayif, "Compiler-directed selection of dynamicmemory layouts," in *Proc. 9th Int. Symp. Hardware/Software Codesign*, vol. 45, 2001, pp. 219–224.

[11] W. Abu-Sufah, D. J. Kuck, and H. L. Duncan, "On the performance enhancement of paging systems through program analysis and transformations," *IEEE Trans. Compute.*, vol. C-30, pp. 341–356, May 1981.

[12] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," in *Proc. ACM Symp. Principles Practice Parallel Program.*, 1991, pp. 39–50.

[13] U. Banerjee, "A theory of loop permutations," presented at the Workshop Lang. Compilers Parallel Computer, Urbana, IL, 1989.

[14] G. Memik, M. Kandemir and A. Choudhary, "Design and Evaluation of Smart Disk Architecture for DSS Commercial Workloads". In Proc. International Conference on Parallel Processing, September 2000.

[15] E. Riedel, C. Faloutsos, G. Gibson and D. Nagle, "Active Disks for Large-Scale Data Processing". IEEE Computer, June 2001, pp. 68–74.

[16] Mayur Naik , Jens Palsberg "Compiling with Code-Size Constraints "ACM Transactions on Embedded Computing Systems, Vol. 3, No. 1, February 2004, Pp 163–181.
[17] S. Udaykumaran and R Barua, "Compiler decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems", CASES'03, San Jose, California, USA, pp. 276-286.
[18] O. Avissar, R. Barua and D Steart , " Heterogeneous Memory Management for Embedded Systems ", CASES'01, 2001, Atlanta, Georgia, USA, pp. 34-43
[19] P.R.Panda, N.D.Dutt, and A. Nicolau, "On-Chip vs Off-Chip Memory : The Data Partitioning Problem in Embedded-Processor based systems", ACM Transaction on Design Automation of Electronic Systems,5(3), July 2000
[20] J. Sjodin, B. Froderberg, and T. Lindgern, "Allocation of Global Data Objects in On-Chip RAM", Compiler and Architecture Support for Embedded Compution Systems, December 1998.
[21] R. Rabbah , "Design Space Exploration of Embedded Memory Systems", Ph. D Thesis, Gorgia Institute of Technology, August 2006.
[22] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proc. ACM Symp. Principles Practice Program. Lang.*, 1988, pp. 319–329.
[23] W. Li and K. Pingali, "A singular loop transformation framework based on nonsingular matrices," *Int. J. Parallel Program.*, vol. 22, no. 2, pp. 183–205, Apr. 1991.
[24] M. Kandemir, J. Ramanujam, and A. Choudhary, "A compiler algorithm for optimizing locality in loop nests," presented at the ACM Int. Conf. Supercomput., Vienna, Austria, 1997.
[25] M. Masmano, I Ripoll and A. Crespo, "Dynamic storage allocation for Real-Time Embedded systems", Spain, 2001
[26]. M. Uysal, A. Acharya and J. Saltz, "Evaluation of Active Disks for Decision Support Databases". In Proc. International Conference on High Performance Computing Architecture, January 2000.
[27] M. Wolfe , "How compilers and tools differ for embedded systems", STMicroelectronics, Inc., 2005.
[28] S.-T. Leung and J. Zahorjan, "Optimizing data locality by array restructuring,"Univ. Washington, Comput. Sci. Dept., Seattle, WA, 1995.
[29] M. O'Boyle and P. Knijnenburg, "Nonsingular data transformations: definition, validity, and application," presented at the Int. Conf. Supercomput., Vienna, Austria, 1997.
[30] Lerie Kane , "Creating High Performance Embedded Applications Through Compiler Optimizations ", Technology@Intel Magazine , march 2005.
[31] Kevin Tucker , "Compiler Optimization And Its Impact On Development Of Real-Time Systems" , DOC-I, Inc., Phoenix, AZ
[32] Rainer Leupers, "Compiler Design Issues for Embedded Processors" Aachen University of Technology,July–August 2002
[33] Wayne Wolf, and Mahmut Kandemir , "Memory System Optimization of Embedded Software " , Proceedings Of The Ieee, Vol. 91, No. 1, January 2003 pp 165-182