Designing and Tuning the Memory Subsystem to Optimize SoC Performance

When optimizing a design, system-on-chip (SoC) designers must balance system performance, processor area (cost), and memory size, typically by tuning a variety of memory subsystem parameters for each on-chip processor. The processor's instruction set simulator (ISS) plays a key role in this process because the ISS can model and quickly report the expected system performance showing a breakdown of memory-related parameters.

In this white paper, we discuss issues and key parameters that will help your SoC design team tune your memory subsystem to yield optimum performance for your target application.

Contents

Introduction

An SoC's memory subsystem architecture is custom-designed and tailored to meet the needs of each target application using a range of implementation options. To help SoC architects, hardware designers, and programmers develop a memory subsystem design optimal for their application, a processor's architecture should have native hardware support for optimizing the interface with the memory subsystem, and the instruction set architecture (ISA) should support software programmability to exploit hardware features that improve application performance.

The memory subsystem design occurs in two phases: first understanding the target application's memory requirements and determining the required memory subsystem, then simulating, measuring memory and cache transactions, and tuning the memory subsystem.

This white paper discusses issues and key parameters for your SoC design team to investigate as you design and tune your memory subsystem for optimum performance.

Architecting the Memory Subsystem

When architecting the memory subsystem, designers must address a variety of basic design issues, such as how instructions and data will be stored, and the type of memory-bus interface required for data transactions. This section provides a number of questions design teams can use to identify and resolve these issues.

Instruction and Data Storage

- **Memory interface:** Will the processor execute its initialization code from off-chip, read-only, or flash memory, or will that code be preloaded into a local on-chip instruction RAM? The answer to this question is important because the boot-code location helps determine the type of memory-bus interface required. If the processor never needs remote memory access except for initialization code, a simpler and smaller memory interface may be appropriate.
- **Instruction cache:** Is the performance-sensitive application code small enough to fit entirely in the processor's local instruction RAM or is an instruction cache necessary? Many design teams automatically assume that cache will be needed, but not all deeply embedded SOC designs require it.
- **Data input:** How will the processor load application input data? Will the run-time data requirements, such as heap and stack, fit into the local data RAM? For example:
  - Is it loaded from remote input buffers or memories?
  - Is it pushed into the processor's local data RAM by an outside agent such as a direct memory access (DMA) controller or another processor?
  - Does the processor directly load the data using input or load instructions?

Memory-Bus Interface Requirements

- **Output data:** How does the output data get to other parts of the SOC? For example:
  - Is it sent to remote output buffers or memories?
  - Is it pulled from the processor's local data RAM by an outside agent such as a DMA controller or another processor?
  - Does the processor transfer the data through direct output operations using output or store instructions?
- **Local data RAM:**
  - Can all of the performance-sensitive data (including application data, a maximum-sized stack, and other variable data storage) fit entirely in the processor's local data RAM?
  - Will it be necessary to emulate a large local data RAM using a local data cache and a large external memory?

Sometimes, an application's instruction or data segments are so large that instruction and data caches are required to achieve good overall system performance. Unfortunately, complex interactions among memory references can make cache behavior appear non-deterministic. This trait presents a significant problem for some embedded applications, where the processor must be able to access certain instruction sequences or data regions with a small, constant latency.

You can address this need either by closely coupling both cache and local RAM to the processor (with time-critical instructions or data allocated to addresses mapped to the local RAM) or by locking certain cache regions to prevent particular data or instruction lines from being evicted from the cache once the correct contents are loaded. Cache-locking temporarily makes selected cache regions act like local memory (on a line-by-line basis).

Measuring Transactions and Tuning the Memory Subsystem

Once you have established the basic organization of instruction, data, and cache memories, you can proceed with the second phase of the memory subsystem configuration: detailed memory subsystem tuning using instruction-set simulation. Analyzing memory stalls associated with the processor's memory subsystem configurations drives this tuning process.

Profiling a configuration can provide valuable information to help you tune your memory subsystem architecture, including total executed instructions, data read instructions executed, data write instructions executed, committed instructions, uncached instructions, instruction-cache misses, data-cache-load misses, store-buffer stalls (for a write-through cache), and other instruction execution delays (such as exceptions, source interlocks, and branch-taken delays), cycles per instruction (CPI), and cumulative CPI.

Using a software tool chain that includes an instruction set simulator (ISS) to profile the configuration allows you to simulate and determine the best memory subsystem for the application early in an SoC's development cycle. The software tool chain lets you change the simulation parameters and gather statistics across numerous settings to find the optimum parameter values.

You can investigate your memory subsystem from two perspectives during the tuning process:

- The **macro** view of the memory subsystem's aggregate performance across all instruction and data references in a complete application
- The **micro** view of the memory subsystem's behavior—especially data references—in the key application hot spots or critical inner loops

Aggregate Memory Subsystem Performance

The macro view includes accumulated dynamic statistics of all program-memory references. These cumulative statistics often give little insight into why, for example, cache misses occur, but they serve as the foundation for tuning overall application throughput. Simulating the application with a range of different memory subsystem parameters establishes the tradeoffs between application performance and memory subsystem implementation cost.

Typically, the behavioral patterns of an application's instruction and data references are uncorrelated—the statistics of each must be examined to determine the appropriate configuration. The impact of various parameters is summarized below:

- **Memory-access latency** (time to the first word): Reducing the number of cycles needed to retrieve the first word of a cache-refill block always reduces the penalty for cache misses and increases application performance.
- **Memory-access time** (time to access each additional word in the cache-refill block): Reducing the incremental delay for each subsequent word in a cache-refill block reduces the penalty for cache misses and increases application performance, but only when the cache-line size is larger than one memory word.
- **Cache size:** Increasing the cache size almost always reduces the number of cache misses, which increases application performance.
- **N-way set associative cache:** Increasing a set associativity cache's number of ways almost always reduces the number of cache misses, which increases application performance.
- **Cache-line size:** Increasing the cache-line size may increase or decrease application performance, depending on memory-reference patterns. Longer cache lines take more time to fill from or write back to main memory, and thus increase the risk of interference among cache lines. When all the words in the cache line are used before a line is evicted from the cache, a longer line size generally improves application performance, particularly when the latency to access the first memory word is long and the incremental time for additional words is short. Conversely, if the application uses little of each cache line before the line is evicted, a shorter cache line may improve application performance, particularly if the memory latency is short or the cost of each incremental word of the cache line is large. This complexity and uncertainty makes accurate simulation critical, as it's often difficult to forecast the optimum configuration.
- **Prefetch:** Long memory latency effects can be reduced by prefetching data or instructions through a variety of methods:
  - **Hardware** prefetch automatically detects when a string of sequential data or instructions are being brought into the data or instruction cache and then requests the next set of data or instructions before they are needed.
  - **Software** prefetch instructs the processor to prefetch individual cache lines when it is known that the data will be needed later.
  - **Block** prefetch allows the programmer to specify an entire range of cache lines to be requested in the background.
- **Refill width:** Increasing the number of bits that can be transferred into or out of the cache on each cycle generally reduces the delay for reading or writing a cache line, therefore improving application performance, especially for long cache lines. The line-refill size often corresponds to the processor's memory-bus width, so instruction- and data-refill widths are usually the same.

- **Write-back vs. write-through data cache:** Choosing a write-back cache generally reduces the number of write operations to the non-local (on-chip or off-chip) memory associated with the data cache because several processor store operations to locations within one cache line result in just one write operation on the processor's main bus. If the bus bandwidth to main memory is relatively constricted, choosing a write-back policy may increase application performance. If write bandwidth is not an issue, a write-through cache can reduce the delay incurred by data-cache misses, which will also increase application performance. The data-cache miss delay is often less for write-through caches because making room for a new cache line (by evicting an existing line) never requires dirty data to be written back. In addition, write-back caches can stall the processor during a write operation if the target cache line is not already in the cache. The cache often must load all of the other words in the target cache line before the write can occur. This type of stall doesn't happen with write-through caches.

Each of these memory-system configuration choices affects the silicon cost of the processor implementation. Larger caches, wider refill width, shorter cache-line size, and greater set associativity all increase the silicon area for RAM, processor logic, or both. Of these choices, increasing cache capacity and decreasing the cache-line size generally incurs the biggest silicon area increases.

Main-memory latency is often a central concern in processor configuration. Memory caches reduce that sensitivity, sometimes dramatically. In many cases, however, optimizing a system's main memory, independent of the processor's local-memory configuration, can be crucial to improving overall performance. For example, accessing off-chip dynamic RAM can often require 50 to 100 processor cycles per access, particularly if the path from processor to memory winds through several different on-chip buses or if the RAM interface is not optimized for rapid access times.

Figure 1 shows a memory hierarchy with memory caches and local RAM, global on-chip RAM, and off-chip RAM.

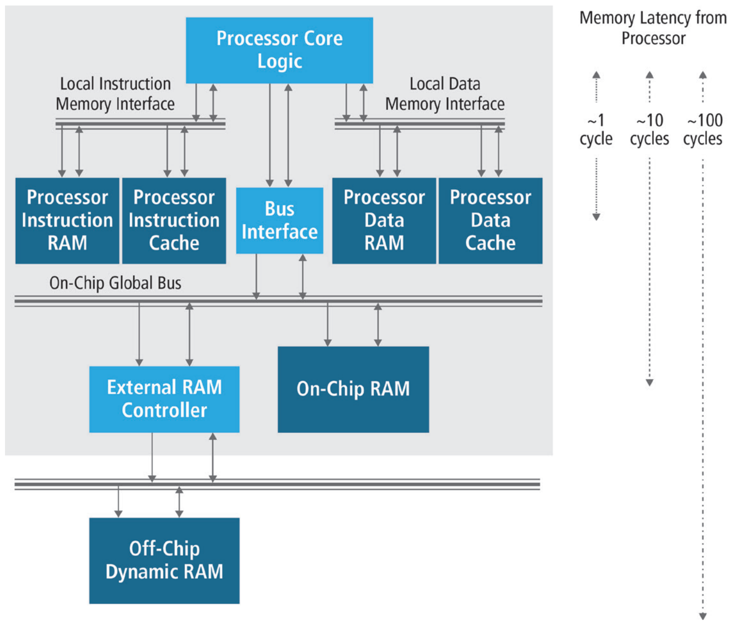

Figure 1: SoC memory hierarchy

The access latency from the processor increases roughly tenfold for each level in the hierarchy:

- Local memory access latency: ~ 1 cycle
- Global on-chip RAM access latency: ~ 10 cycles
- Off-chip RAM access latency: ~ 100 cycles

Configurable Processors and Memory Hierarchy

Configurable processors enable the designer to adjust many different memory-system parameters, including the ones listed above. Understanding the performance tradeoffs and choosing the optimal configuration for all of these memory parameters is a complex process. Charts of application performance as a function of key parameters can help you visualize those tradeoffs and finalize the memory subsystem more quickly.

Figure 2 shows simulated data-cache performance results for an optimized JPEG encoding application. It plots the total number of execution cycles as a function of cache size, cache-line size, and set associativity.

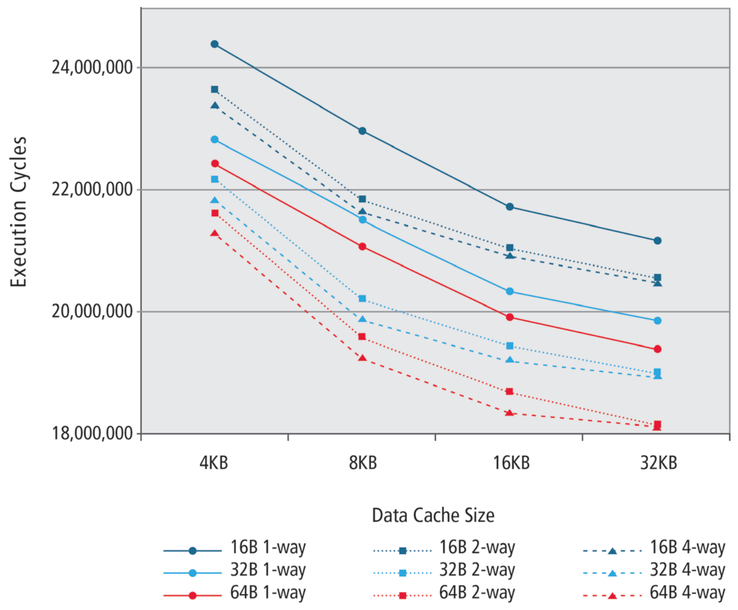

Figure 2: Data cache size performance results

Data-cache behavior is an important design consideration in this application. The simplest cache (4KB, direct mapped, 16-byte line) has a load miss rate of 13.4%, while the most complex cache (32KB, 4-way set associative, 64-byte line) has a load miss rate of 1.9%. Figure 2 clearly shows that larger cache sizes are better, but it also suggests that you'll see diminishing returns for cache sizes greater than 16KB.

Cache-line size—the number of bytes brought in for each cache miss—is also a significant factor in this example. Increasing the cache-line size from 16 to 64 bytes creates more performance benefit than doubling the cache size. This result is important because longer cache lines actually reduce silicon area and cost, while doubling cache size can be expensive. Figure 2 also shows that two-way set associativity is clearly better than direct-mapped (one-way) cache, but going from two-way to four-way set associativity has less dramatic benefits.

The simulation results shown above are based on a single-processor design. No consideration was made for other processors that might be contending for access to shared, non-local memory. Processor behavior can change in multiprocessor systems depending on the memory design, the interconnect structure between processors and memory, and the pattern of memory references by other processors in the system.

Innovate with Cadence Tensilica Processors

Whether you focus on one product or create thousands, requirements change over time. Those changes often include reducing cost, increasing performance, and/or increasing energy efficiency. Using a Cadence® Tensilica® processor and its software development tools, you can meet these new requirements with less cost and risk than other approaches.

Tensilica processors offer a single instruction set architecture and set of development tools that scale from tiny microcontrollers and digital signal controllers to high-performance real-time controllers and DSPs. And, using a higher performance or more functional processor does not require learning a new set of software development tools or a new hardware flow—the same flow is used across all Tensilica processors.

Easy Customization and Optimization

Tensilica processors can be easily optimized to run your application more efficiently. Choose to add instructions to the base instruction set from pre-defined options or create your own using the Verilog-like Tensilica Instruction Extension (TIE) language. You can add elements to include deep data pipelines, parallel execution units, task-specific state registers, wide data buses to local and global memories, and virtually unlimited I/Os. The processor is customized with an automated flow that requires no extra processor verification and keeps the development tools updated with every change—guaranteeing that your new processor is correct-by-construction.

Keep programmability in your designs even when a conventional fixed-ISA processor needs to be offloaded because the instruction set is unsuitable or I/O-limited. Using Tensilica processors, you can build the "offload" with more application-specific instructions and I/O in a fraction of the time that it would take to design and verify in RTL. You can also avoid any interfacing delays that may result from implementing this functionality in an external logic block—new instructions can use the general and/or custom registers sized to the data types involved for instant availability.

Optimizing the processor with new instructions and I/O creates a unique design, making your particular application code run much more efficiently—reveal them to your own programmers or your customer's programmers only as needed.

The development tools are created automatically to handle all optimizations. Full C/C++ support is provided with all the debug and profiling that you expect (and need) for professional development with our familiar Eclipse-based integrated design environment (IDE).

Connecting Memories to the Tensilica Processor

The Tensilica processor offers a variety of options for connecting memories, such as:

- Simultaneous connections to local interfaces for data and instruction cache RAMs, tag RAM, other RAMs, and data and instruction ROMs—with a 1-cycle latency for a 5-stage pipeline and 2-cycle latency for a 7-stage pipeline
- Add unlimited direct memory connections that are independent of local and system bus transfers through one or more direct Lookup interfaces (for tables and scratchpad memory)—with a fixed latency
- System bus interface for access to global memories—latency dependent on bus protocols and activity

Tensilica Processor Software Tools

Cadence provides the easy-to-use, Eclipse-based Xtensa® Xplorer automation tools, a visual environment that enables designers to develop various configurations and makes creating Tensilica processor-based SoC hardware and software design much easier. All software development tools—including compilers, debuggers, ISS, and compilers—are provided. Xtensa Xplorer integrates software development, processor optimization, and multiple-processor SoC architecture tools, as well as SoC simulation and analysis tools, into one common design environment. Using Xtensa Xplorer you can create, program, and debug shared memory multiprocessor subsystems, and simultaneously develop code on more than one processor configuration.

Architect the optimal memory subsystem configuration for your SoC design by gathering values for key memory parameters, as discussed earlier. Iterate various design configurations and tune the memory subsystem using the ISS for profiling and linker support packages (LSPs) for code and data placement until the memory subsystem configuration meets your target application's requirements.

Reducing the Memory Footprint

You can reduce the binary code size of your software application code running on Tensilica processors using a variety of methods, enabling you to reduce application execution times and your processor memory requirements—and improving processing efficiency. Refer to the Reducing Your Code Size on Xtensa Processors Application Note for details.

Conclusion

When designing a memory subsystem for a specific application, design teams are challenged to evaluate a number of factors to find the right balance between performance and latency. The memory subsystem has a dramatic impact on your application's performance and on the cost of the system. Simulating your application in the ISS will help you evaluate tradeoffs when choosing the processor's local memories, caches, and bus widths as well as when tuning the rest of the memory subsystem.

Consider using Tensilica processors for the ultimate flexibility to design an optimum memory subsystem that is best suited for your application.

Additional Information

For more information on the unique abilities and features of Cadence Tensilica processors, see ip.cadence.com.