Automated Software Cache Management

William Lundgren (wlundgren@gedae.com), Kerry Barnes (kbarnes@gedae.com), James Steed (jsteed@gedae.com)
Gedae, Inc., 1247 N. Church St., Suite 5, Moorestown, NJ 08057

Introduction

Hardware cache management is used in a majority of chip designs (including major Intel architecture and Power architecture chips) to manage the movement of data and instructions from remote slow memory to local fast memory. Hardware cache management is heuristic and cannot optimally manage movement for all applications. Additionally it takes up chip real estate that could be used for floating point operations (FLOPs). Several recent chip designs have radically removed the hardware cache manager, including the Cell Broadband Engine (Cell/B.E.) processor and ClearSpeed CSX700, using scratchpad memory instead of cache and devoting real estate to more cores and ALU paths at the expense of ease of programming. The size limitations for on-chip scratchpad memory make large programs difficult to implement and necessitate orchestrating the movement of both data and instructions (overlays) from off-chip memory to on-chip memory in the code. Both approaches to cache management are flawed. We propose moving cache management from the burden of the programmer or the burden of the hardware to the burden of the compiler. Synthetic Aperture Radar (SAR) benchmarks are provided to illustrate performance improvement using the Gedae compiler on hardware cache architectures and its use as an enabling technology for scratchpad memory architectures.

Data Caching

Large data sets can often be decomposed to make better use of on-chip memory, often referred to as stripmining the data. In scratchpad memory architectures, the application may not be possible unless the data is stripmined in small enough portions that the data fits in local memory. In hardware cache architectures, the cache usually works in concert with virtual memory to guarantee correct behavior, however it is not guaranteed to be efficient.

As a case study, consider the implementation of the signal processing portion of the SAR benchmark. This SAR algorithm has two key stages: the range processing of the rows of the matrix and the azimuth processing of the columns of the matrix. To distribute the algorithm a distributed transpose – or corner turn – is performed between the two stages to redistribute the data. If the input image is size RxC, the range processing processes vectors of size C, and the azimuth processing processes vectors of size 2*R. The algorithm can be coded as

```
r_in[j](i) = in[i][j];//mat-to-vec
r_out[j] = range(r_in);
az_in[i](j) = r_out[j](i);//cturn
az_out[i2] = azimuth(az_in);
out[j][i] = az_out[i](j);//select
```

For this benchmark, the Gedae language exposes the situations where stripmining can be applied. The range and azimuth stages are specified as rowwise operates that naturally can be processed N rows at a time.

An experiment was performed to port the SAR benchmark to a multiprocesor VMX board and determine the speedup between four core, eight core, and sixteen core systems. The VMX board is based on the FreeScale PowerPC 8641D processor which has 1MB L2 cache for both data and instructions. Processing a 512x2048 complex matrix in all cases, code with no software cache optimization achieves superlinear speedup. Because the image size stayed constant, the workload granularity decreases as the number of processors increases. The only difference between the implementations is the number of vectors processed at a time; the application processes 8/P MB in the range processing and 16/P MB in the azimuth processing where P is the number of cores. The 16 core implementation achieves better performance because there is less reliance on the hardware to manage the cache.

The Gedae compiler can compensate for the size of cache and adjust the implementation to better stripmine the data. Using automated cache management, a 72% improvement is achieved on the 4 core implementation, and all implementations are improved. Performance vs. the number of cores is now linear.

Table 1 – Performance Improvement in 512x2048 SAR with Automated Software Cache Management

	4 cores	*8 cores*	*16 cores*
Default	0.205 s	0.0939 s	0.0354 s
Gedae	0.119 s	0.0624 s	0.0315 s
% Diff	72%	50%	12%

The same automated stripmining capability can be used to enable porting applications to architectures that use scratchpad memory. The compiler stripmines data and insert DMA calls into the application. The compiler is capable of handling all the careful data flow planning and organization that must be done to make sure data arrives just as it is ready to be processed, overlapping communication and processing with double buffering to maintain throughput. Performance results on the Cell/B.E. architecture have been reported in [1] and [4].

Instruction Caching

Large program sizes can have a similar effect on programmability of scratchpad memory based architectures. The size of the scratchpad memory on a ClearSpeed CSX700 is 128 kB and on a Cell/B.E.'s SPE is 256 kB.

With such modest sized local memory and competing high demands for data storage in the same memory, programs are often decomposed temporally into code overlays. While the problem is more profound for scatchpad based architectures, potentially performance benefits can be achieved on hardware cache architectures if instructions are more intelligently decomposed into overlays.

The Gedae compiler provides a method for automatically implementing code overlays for an application. The capability has been demonstrated on the Cell/B.E. architecture and is being used on a production program. The compiler forms threads by scheduling the execution of kernels at compile time. Overlays are formed by decomposing those thread schedules into subschedules and building the subschedule into an overlay. Each kernel is built into separate object files, allowing overlays to be created from collections of kernels. Subschedules are identified by analyzing the application for stripmining opportunities; if data is stripmined, there is a portion of code that is reused several times before being discarded, forming a natural overlay. Additionally, thread schedules can be decomposed temporarily (like pipeline stages) to create overlays.

The overlays form a tree structure where the root of the tree is retained in memory, and each node below the root represents a subschedule of the parent node. The Gedae language provides knowledge to the compiler which makes the generation of this tree structure possible, including where each pointer is used and how it is used (read, write, or both). The compiler is able to manage state information more effectively using the tree model. The compiler marks all state data that must be retained between overlay executions and manages the storage and retrieval of state data in system memory during context switches. As the tree is descended at runtime, data from parent overlays that is used in child overlays can be retained. As the tree is ascended, data that is no longer relevant is easily discarded. Additionally, this tree structure includes conditional execution; if data can be branched to N different paths, only the code for the path that is followed is downloaded.

All overlays are built at compile time and loaded into system memory at program initialization, thus the total time for a context switch is the time to put (upload) the preceding overlay's state information into system memory and get (download) the next overlay's object into the destination region in the on-chip memory. The tree structure also provides the opportunity to double buffer the loading of code if the on chip memory is sufficiently large. Aside from the bandwidth limitations of the system memory controller (25.6 GB/s on the Cell/B.E. architecture (see [1] for a discussion), 4 GB/s on the ClearSpeed architecture), there is very little overhead in an overlay context switch. An overlay manager is included in the SPE image, and all overlay control resides on the SPEs. A context switch between the three stages of the SAR algorithm (range, corner turn, azimuth) takes approximately 40 µs for all 8 SPEs, where the overlays range in size from 42 to 64 kB. (A context switch to a small overlay can be as small as 1 µs.) This time includes several small transfers (putting state data) and one large transfer (getting the new overlay), and there may be competition over the system memory controller as other processors get and put data.

As a comparison, IBM's Cell Software Development Kit (Cell SDK) includes the capability to create overlays through the creation of linker scripts. Similar to the Gedae compiler's capability, the linker script allows for the creation of a tree structure of overlays where the root of the tree remains resident in memory. The Cell SDK's overlay capability does not manage state; all data not in the main overlay is considered transient. The IBM XLC compiler can automate the creation of these overlays.

The overhead of the Cell SDK overlay context switch is high. Included in the Cell SDK are several overlay examples. The most substantial is a large matrix processing example. The example uses two overlays, one sized 20187 B and one sized 2276 B. The example can be run with and without overlays. The runtime with overlays is 11.26 ms, and the runtime without overlays is 10.75 ms. For this example, the cumulative overlay context switch time is 0.51 ms for a single SPE on a modest sized overlay. Data is only retrieved (no state is stored), and there is no other competition for the memory bandwidth in this example as all program data resides in local storage.

Conclusions

The Gedae compiler is able to automatically manage data and instruction caching to both enable programming of scratchpad based architectures and optimize processing on hardware cache based architectures. The automated code overlay capability provided by the Gedae compiler provides better performance (on the order of 10,000-50,000x) and more flexibility than competing solutions. Cache management can be jointly optimized based on the application structure and the target architecture, and the optimizations are done at compile time eliminating almost if not all of the runtime overhead. As a result, applications which were previously unthinkable on processors with lightweight cores are now easily generatable from Gedae. That is, applications can now be automatically decomposed into overlays small enough for lightweight cores with minimal additional effort from the programmer. Even for processors that use hardware cache, Gedae provides better cache locality and therefore runs more efficiently.

References

[1] Barnes, K. et al. "Implementation of 2-D FFT on the Cell Broadband Engine Architecture," HPEC, 2009.

[2] Curtiss Wright Controls. "The CHAMP-AV6 VPX-REDI Digital Signal Processing Card - Maximizing Performance with Minimal Porting Effort," 2010. <http://www.cwcembedded.com>.

[3] IBM. *Software Development Kit for Multicore Acceleration Version 3.0 Programmer's Guide*, SC33-8325-02, 2007. <http://www.ibm.com>.

[4] Lundgren, W. et al. "Simple, Efficient, Portable Decomposition of Large Data Sets," HPEC, 2008.